TUGAS AKHIR - KI141502

DESAIN DAN ANALISIS ALGORITMA KOMPUTASI JUMLAH BRIDGE PADA GRAF DINAMIS INKREMENTAL

YUSRO TSAQOVA
NRP 5112 100 095

Dosen Pembimbing 1
Bilqis Amaliah, S.Kom., M.Kom.

Dosen Pembimbing 2
Rully Soelaiman, S.Kom., M.Kom.

JURUSAN TEKNIK INFORMATIKA
Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember
Surabaya, 2016

UNDERGRADUATE THESES - KI141502

ALGORITHM DESIGN AND ANALYSIS FOR BRIDGE COMPUTATION ON INCREMENTAL DYNAMIC GRAPH

YUSRO TSAQOVA
NRP 5112 100 095

Supervisor 1
Bilqis Amaliah, S.Kom., M.Kom.

Supervisor 2
Rully Soelaiman, S.Kom., M.Kom.

INFORMATICS DEPARTMENT
Faculty of Information Technology
Institut Teknologi Sepuluh Nopember
Surabaya, 2016

LEMBAR PENGESAHAN

DESAIN DAN ANALISIS ALGORITMA KOMPUTASI JUMLAH BRIDGE PADA GRAF DINAMIS INKREMENTAL

TUGAS AKHIR

Diajukan Untuk Memenuhi Salah Satu Syarat
Memperoleh Gelar Sarjana Komputer
pada
Rumpun Mata Kuliah Dasar dan Terapan Komputasi
Program Studi S-1 Jurusan Teknik Informatika
Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember

Oleh:

Yusro Tsaqova
NRP. 5112 100 095

Disetujui oleh Pembimbing Tugas Akhir:

Bilqis Amaliah, S.Kom, M.Kom. .....
NIP. 197509172001122002 (Pembimbing 1)

Rully Soelaiman, S.Kom., M.Kom .....
NIP. 197002131994021001 (Pembimbing 2)

SURABAYA
JANUARI 2016

DESAIN DAN ANALISIS ALGORITMA KOMPUTASI JUMLAH BRIDGE PADA GRAF DINAMIS INKREMENTAL

Nama	: YUSRO TSAQOVA
NRP	: 5112 100 095
Jurusan	: Teknik Informatika FTIF - ITS
Pembimbing I	: Bilqis Amaliah, S.Kom., M.Kom.
Pembimbing II	: Rully Soelaiman, S.Kom., M.Kom.

Abstrak

Komputasi jumlah bridge pada graf dinamis inkremental merupakan permasalahan perhitungan jumlah bridge pada sebuah graf yang bentuknya berubah secara dinamis karena pertambahan edge. Untuk mendapatkan jumlah bridge pada graf bisa dilakukan dengan komputasi ulang untuk setiap operasi penambahan edge namun pendekatan tersebut tentunya tidak efisien.

Pada Tugas Akhir ini akan dirancang penyelesaian permasalahan di atas dengan melihat beberapa kondisi yang harus diperhatikan dari pasangan vertex yang mengalami pertambahan edge yaitu apakah pasangan vertex berada dalam himpunan connected-component yang berbeda, himpunan connected-component yang sama namun dalam himpunan bridge-block yang berbeda atau dalam himpunan bridge-block yang sama. Implementasi dari solusi ini menggunakan bantuan struktur data disjoint set ditambah pendekatan heuristik union by weight dan path compression.

Hasil dari Tugas Akhir ini telah berhasil menyelesaikan permasalahan di atas dengan cukup efisien dengan kompleksitas waktu $\mathcal{O}(M(\alpha(N)))$ *dimana* $\alpha(N)$ *adalah invers dari fast-growing ackermann function dan bernilai* ≤ 4 *untuk setiap N.*

Kata Kunci: Graf, Bridge, Online, Pencarian, Dinamis, Inkremental

Halaman ini sengaja dikosongkan

ALGORITHM DESIGN AND ANALYSIS FOR BRIDGE COMPUTATION ON INCREMENTAL DYNAMIC GRAPH

Name	: YUSRO TSAQOVA
NRP	: 5112 100 095
Major	: Informatics Department Faculty of IT - ITS
Supervisor I	: Bilqis Amaliah, S.Kom., M.Kom.
Supervisor II	: Rully Soelaiman, S.Kom., M.Kom.

Abstract

Bridge number computation on incremental graph is a problem of computation of bridge number on a graph which is dynamically changed because of edge addition. For obtaining bridge total number in a graph, the use of re-computation for every single operation of edge addition is possible, but certainly it is inefficient.

This undergraduate thesis will design the problem solving of the problem above by seeing some conditions which should be considered from vertex pair which has edge addition, which is whether vertex are located in same connected-component set or not but in a same bridge-block set or in similar bridge-block set. The implementation of this solution is using the help of disjoint set data structure and heuristic union by weight and path compression.

The result of this undergraduate thesis has successfully solved the problem mentioned with sufficient enough by the $\mathcal{O}(M(\alpha(N)))$ time complexity where $\alpha(N)$ is the inverse of fast-growing ackermann function and has less than or equal 4 for every single N.

Kata Kunci: Graph, Bridge, Online, Searching, Dynamic, Incremental

Halaman ini sengaja dikosongkan

KATA PENGANTAR

Puji syukur penulis panjatkan kepada Allah SWT atas segala rahmat dan karunia-Nya sehingga memungkinkan penulis untuk dapat menyelesaikan Tugas Akhir yang berjudul:

DESAIN DAN ANALISIS ALGORITMA KOMPUTASI JUMLAH *BRIDGE* PADA GRAF DINAMIS INKREMENTAL.

Penelitian Tugas Akhir ini dilakukan untuk memenuhi salah satu syarat meraih gelar Sarjana di Jurusan Teknik Informatika Fakultas Teknologi Informasi Institut Teknologi Sepuluh Nopember.

Dengan selesainya Tugas Akhir ini diharapkan apa yang telah dikerjakan penulis dapat memberikan manfaat bagi perkembangan ilmu pengetahuan terutama di bidang teknologi informasi serta bagi diri penulis sendiri selaku peneliti.

Penulis mengucapkan terima kasih kepada semua pihak yang telah memberikan dukungan baik secara langsung maupun tidak langsung selama penulis mengerjakan Tugas Akhir maupun selama menempuh masa studi antara lain:

- Allah SWT atas segala rahmat dan karunia yang telah diberikan selama ini.
- Ayah, Ibu dan keluarga penulis yang selalu memberikan perhatian, dorongan dan kasih sayang yang menjadi semangat utama bagi diri penulis baik selama penulis menempuh masa kuliah maupun pengerjaan Tugas Akhir ini.
- Bapak Rully Soelaiman, S.Kom., M.Kom. selaku Dosen Pembimbing yang telah banyak meluangkan waktu untuk memberikan ilmu, nasihat, motivasi, pandangan dan bim-

bingan kepada penulis baik selama penulis menempuh masa kuliah maupun selama pengerjaan Tugas Akhir ini.

- Ibu Bilqis Amaliah, S.Kom., M.Kom. selaku Dosen Pembimbing yang telah memberikan dukungan, nasihat, arahan dan masukan kepada penulis.
- Seluruh tenaga pengajar dan karyawan Jurusan Teknik Informatika ITS yang telah memberikan ilmu dan waktunya demi berlangsungnya kegiatan belajar mengajar di Jurusan Teknik Informatika ITS.
- Teman-teman Laboratorium Pemrograman angkatan 2011, 2012 dan 2013 yang telah menjadi keluarga di kampus.
- Seluruh teman penulis di Jurusan Teknik Informatika ITS yang telah memberikan dukungan dan semangat kepada penulis selama penulis menyelesaikan Tugas Akhir ini.
- Seluruh pihak yang tidak bisa penulis sebutkan satu-persatu yang telah memberikan dukungan selama penulis menyelesaikan Tugas Akhir.

Penulis mohon maaf apabila masih ada kekurangan pada Tugas Akhir ini. Penulis juga mengharapkan kritik dan saran yang membangun untuk pembelajaran dan perbaikan di kemudian hari. Semoga melalui Tugas Akhir ini penulis dapat memberikan kontribusi dan manfaat yang sebaik-baiknya.

Surabaya, Januari 2016

Yusro Tsaqova

DAFTAR ISI

SAMPUL i

LEMBAR PENGESAHAN vii

ABSTRAK ix

ABSTRACT xi

KATA PENGANTAR xiii

DAFTAR ISI xv

DAFTAR TABEL xix

DAFTAR GAMBAR xxi

DAFTAR KODE SUMBER xxv

1 PENDAHULUAN 1

1.1 Latar Belakang 1

1.2 Rumusan Masalah 3

1.3 Batasan Masalah 3

1.4 Tujuan 3

1.5 Metodologi 4

1.6 Sistematika Penulisan 5

2 DASAR TEORI 7

2.1 Deskripsi Permasalahan 7

2.2 Strategi Penyelesaian Permasalahan 9

2.3 *Disjoint Set* 13

2.4 Lowest Common Ancestor (LCA) 19
2.5 Permasalahan *Online Bridge Searching* pada SPOJ 22
2.6 Pembuatan Data Generator untuk Uji Coba 29

3 DESAIN **33**
3.1 Deskripsi Umum Sistem 33
3.2 Desain Kelas *Node* dan Fungsi Init 34
3.3 Desain Fungsi InsertEdge 35
3.4 Desain Fungsi FindBridgeBlock 36
3.5 Desain Fungsi FindComponent 36
3.6 Desain Fungsi SetRoot 37
3.7 Desain Fungsi MergeBridgeBlock 38

4 IMPLEMENTASI **39**
4.1 Lingkungan Implementasi 39
4.2 Implementasi Fungsi Main 39
4.3 Implementasi Variabel Global 40
4.4 Implementasi Struct *Node* 41
4.5 Implementasi Fungsi InsertEdge 42
4.6 Implementasi Fungsi FindBridgeBlock 43
4.7 Implementasi Fungsi FindComponent 44
4.8 Implementasi Fungsi SetRoot 44
4.9 Implementasi Fungsi MergeBridgeBlock 45

5 UJI COBA DAN EVALUASI **47**
5.1 Lingkungan Uji Coba 47
5.2 Skenario Uji Coba 47
5.2.1 Uji Coba Kebenaran 47
5.2.2 Uji Coba Kinerja 58

6 KESIMPULAN **63**
6.1 Kesimpulan 63

DAFTAR PUSTAKA **65**

Lampiran A Hasil uji pada situs SPOJ sebanyak 30 kali 67

Lampiran B Hasil uji kinerja jumlah operasi M tetap 71

Lampiran C Hasil uji kinerja jumlah *vertex* N tetap 79

BIODATA PENULIS 87

Halaman ini sengaja dikosongkan

DAFTAR GAMBAR

2.1 Ilustrasi permasalahan. 8
2.2 Ilustrasi komputasi jumlah *bridge* menggunakan algoritma *Bridge-Finding*. 10
2.3 Ilustrasi *bridge*, dan *bridge-block*. 11
2.4 Ilustrasi penambahan *edge*(u, v) dengan u dan v berada pada *connected component* yang berbeda. . . . 11
2.5 Ilustrasi penambahan *edge*(u, v) dengan u dan v berada pada *bridge-block* yang berbeda namun dalam *connected component* yang sama. 12
2.6 *Path-compression* pada *disjoint set*. 14
2.7 Ilustrasi *disjoint set* dari graf pada permasalahan. . 15
2.8 Ilustrasi penambahan *bridge* dan representasi pada *disjoint set* (1). 16
2.9 Ilustrasi penambahan *bridge* dan representasi pada *disjoint set* (2). 17
2.10 Ilustrasi penambahan *bridge* dan representasi pada *disjoint set* (3). 17
2.11 Ilustrasi penambahan *bridge* dan representasi pada *disjoint set* (4). 18
2.12 Ilustrasi penambahan *bridge* dan representasi pada *disjoint set* (5). 18
2.13 Contoh ilustrasi LCA dari 2 *vertex*. 20
2.14 Ilustrasi *cycle* dan penerapan LCA pada permasalahan (1). 21
2.15 Ilustrasi *cycle* dan penerapan LCA pada permasalahan (2). 21
2.16 Deskripsi permasalahan *Online Bridge Searching* di SPOJ. 22

2.17 Contoh masukan dan keluaran pada soal *Online Bridge Searching*. 24
2.18 Ilustrasi operasi yang menghasilkan *bridge* pada contoh permasalahan. 24
2.19 Ilustrasi *disjoint set* pada contoh permasalahan. . . 25
2.20 Ilustrasi operasi yang menggabungkan beberapa *bridge-block* dan menghapus *bridge* pada contoh permasalahan. 26
2.21 Ilustrasi *disjoint set* dan LCA saat pembentukan *cycle* (1). 26
2.22 Ilustrasi *disjoint set* dan LCA saat pembentukan *cycle* (2). 27
2.23 Ilustrasi graf dan *disjoint set* setelah operasi penambahan *edge*(1, 4). 28
2.24 Ilustrasi graf dan *disjoint set* setelah operasi penambahan *edge*(2, 4). 28
2.25 Ilustrasi kondisi akhir dari graf dan *disjoint set* pada contoh kasus *Online Bridge Searching*. 29
2.26 Pseudocode Fungsi Main Generator Skenario Uji 1 30
2.27 Pseudocode Fungsi Main Generator Skenario Uji 2 31

3.1 Pseudocode Fungsi Main 33
3.2 Pseudocode Fungsi Init 34
3.3 Pseudocode Fungsi InsertEdge 35
3.4 Pseudocode Fungsi FindBridgeBlock 36
3.5 Pseudocode Fungsi FindComponent 37
3.6 Pseudocode Fungsi SetRoot 37
3.7 Pseudocode Fungsi MergeBridgeBlock 38

5.1 Contoh kasus uji hasil dari data generator. 48
5.2 Inisialisasi contoh kasus uji. 49
5.3 Ilustrasi penambahan *edge*(5, 3) pada contoh kasus uji. 49

5.4 Ilustrasi penambahan *edge*(5, 4) pada contoh kasus uji. 50
5.5 Ilustrasi penambahan *edge*(1, 4) pada contoh kasus uji. 51
5.6 Ilustrasi penambahan *edge*(2, 5) pada contoh kasus uji. 51
5.7 Ilustrasi penambahan *edge*(4, 2) pada contoh kasus uji (1). 52
5.8 Ilustrasi penambahan *edge*(4, 2) pada contoh kasus uji (2). 53
5.9 Ilustrasi penambahan *edge*(3, 2) pada contoh kasus uji (1). 53
5.10 Ilustrasi penambahan *edge*(3, 2) pada contoh kasus uji (2). 54
5.11 Ilustrasi penambahan *edge*(5, 1) pada contoh kasus uji (1). 55
5.12 Ilustrasi penambahan *edge*(5, 1) pada contoh kasus uji (2). 55
5.13 Ilustrasi penambahan *edge*(1, 3) pada contoh kasus uji (1). 56
5.14 Ilustrasi penambahan *edge*(1, 3) pada contoh kasus uji (2). 56
5.15 Hasil luaran program pada contoh kasus uji. 57
5.16 Hasil uji kebenaran pada situs SPOJ. 57
5.17 Peringkat waktu eksekusi program *Online Bridge Searching* pada situs SPOJ. 58
5.18 Grafik Hasil uji pada situs SPOJ sebanyak 30 kali. . 60
5.19 Grafik hasil uji coba pengaruh jumlah *vertex* terhadap pertumbuhan waktu. 61
5.20 Grafik hasil uji coba pengaruh jumlah operasi terhadap pertumbuhan waktu. 62

A.1 Hasil uji pada situs SPOJ sebanyak 30 kali (1). . . 67

A.2 Hasil uji pada situs SPOJ sebanyak 30 kali (2). . . 68
A.3 Hasil uji pada situs SPOJ sebanyak 30 kali (3). . . 69

DAFTAR KODE SUMBER

4.1 Implementasi Fungsi Main 40
4.2 Implementasi Variabel Global 40
4.3 Implementasi Struct *Node* 41
4.4 Implementasi Fungsi InsertEdge 42
4.5 Implementasi Fungsi FindBridgeBlock 43
4.6 Implementasi Fungsi FindComponent 44
4.7 Implementasi Fungsi SetRoot 44
4.8 Implementasi Fungsi MergeBridgeBlock 45

Halaman ini sengaja dikosongkan

DAFTAR TABEL

B.1 Hasil uji dengan nilai jumlah operasi M tetap 25.000. 71
B.1 Hasil uji dengan nilai jumlah operasi M tetap 25.000 (lanjutan). 72
B.2 Hasil uji dengan nilai jumlah operasi M tetap 50.000. 73
B.2 Hasil uji dengan nilai jumlah operasi M tetap 50.000 (lanjutan). 74
B.3 Hasil uji dengan nilai jumlah operasi M tetap 75.000. 75
B.3 Hasil uji dengan nilai jumlah operasi M tetap 75.000 (lanjutan). 76
B.4 Hasil uji dengan nilai jumlah operasi M tetap 100.000. 77
B.4 Hasil uji dengan nilai jumlah operasi M tetap 100.000 (lanjutan). 78

C.1 Hasil uji dengan nilai jumlah *vertex* N tetap 12.500. 79
C.1 Hasil uji dengan nilai jumlah *vertex* N tetap 12.500 (lanjutan). 80
C.2 Hasil uji dengan nilai jumlah *vertex* N tetap 25.000. 81
C.2 Hasil uji dengan nilai jumlah *vertex* N tetap 25.000 (lanjutan). 82
C.3 Hasil uji dengan nilai jumlah *vertex* N tetap 37.500. 83
C.3 Hasil uji dengan nilai jumlah *vertex* N tetap 37.500 (lanjutan). 84
C.4 Hasil uji dengan nilai jumlah *vertex* N tetap 50.000. 85
C.4 Hasil uji dengan nilai jumlah *vertex* N tetap 50.000 (lanjutan). 86

Halaman ini sengaja dikosongkan

BAB 1

PENDAHULUAN

Pada bab ini akan dijelaskan latar belakang, rumusan masalah, batasan masalah, tujuan, metodologi dan sistematika penulisan Tugas Akhir.

1.1 Latar Belakang

Teori graf adalah salah satu cabang ilmu yang sudah ada sejak lama dan banyak sekali diterapkan untuk membantu pemecahan masalah terutama di bidang ilmu pengetahuan dan teknologi informasi. Banyak permasalahan di dunia nyata seperti komputasi jalur terpendek atau informasi yang terkandung pada hubungan pertemanan di media sosial yang dapat dimodelkan dengan bantuan teori graf untuk selanjutnya dilakukan komputasi dan didapatkan hasil yang diinginkan. Dalam beberapa dekade terakhir tidak sedikit algoritma dan struktur data yang diciptakan untuk mengatasi permasalahan graf.

Graf dinamis adalah sebuah model permasalahan graf dimana informasi yang diinginkan bisa didapat kapan saja dari sebuah graf yang bentuknya berubah-ubah dikarenakan operasi modifikasi berupa penambahan maupun pengurangan *vertex* atau *edge* pada graf tersebut [1]. Permasalahan graf dinamis dapat diklasifikasikan menjadi *fully dynamic* dan *partially dynamic*. Sebuah masalah dikatakan *fully dynamic* apabila dalam masalah tersebut terdapat dua jenis operasi modifikasi yaitu penambahan dan penghapusan *vertex* atau *edge*. Sebuah masalah dikatakan *partially dynamic* apabila hanya ada satu jenis operasi dari penambahan atau penghapusan yang

dapat dilakukan. Jika hanya operasi penambahan yang dapat dilakukan, maka masalah tersebut dikatakan inkremental. Jika hanya operasi penghapusan yang dapat dilakukan, maka masalah tersebut dikatakan dekremental.

Salah satu topik yang cukup menarik pada teori graf adalah *bridge*. *Bridge* pada graf adalah sebuah *edge* yang apabila *edge* tersebut dihapus maka akan meningkatkan jumlah *connected component* pada sebuah graf. Penulis tertarik melakukan penelitian dalam pemecahan masalah yang berhubungan dengan teori graf khususnya permasalahan komputasi jumlah *bridge* pada graf dinamis.

Permasalahan yang diangkat pada Tugas Akhir ini adalah permasalahan komputasi jumlah *bridge* pada graf yang berubah secara dinamis karena operasi penambahan *edge* atau disebut juga graf dinamis inkremental. Diberikan sebuah *undirected graph* yang terdiri dari N *vertex* dan pada awalnya graf tersebut tidak memiliki *edge*. Selanjutnya akan dilakukan M operasi penambahan *undirected edge* yang menghubungkan antara sebuah *vertex* dengan *vertex* lainnya. Untuk setiap operasi penambahan *edge* diperlukan informasi total jumlah *bridge* yang terdapat pada graf tersebut. *Edge* yang ditambahkan untuk setiap operasi dipastikan bukan merupakan *edge* yang telah ada sebelumnya ataupun *edge* yang menghubungkan sebuah *vertex* dengan *vertex* itu sendiri.

Solusi naif dari permasalahan di atas adalah dengan melakukan komputasi ulang jumlah *bridge* untuk setiap operasi penambahan *edge*, namun solusi tersebut sangatlah tidak efisien mengingat jumlah *edge* pada graf terus bertambah karena adanya operasi penambahan *edge*. Oleh karena itu, dibutuhkan desain algoritma dan struktur data yang efisien baik dalam kecepatan komputasi maupun penggunaan memori untuk permasalahan ini.

Hasil dari Tugas Akhir ini diharapkan dapat menentukan implementasi algoritma dan struktur data yang tepat untuk memecahkan per-

masalahan di atas secara optimal dan dapat memberikan kontribusi terhadap perkembangan pengetahuan teknologi informasi.

1.2 Rumusan Masalah

Rumusan masalah yang diangkat dalam Tugas Akhir ini adalah sebagai berikut:

1. Bagaimana menganalisis dan menentukan algoritma dan struktur data yang tepat untuk menyelesaikan permasalahan komputasi jumlah *bridge* pada graf dinamis inkremental dengan optimal.
2. Bagaimana implementasi dari algoritma dan struktur data yang dirancang dalam penyelesaian permasalahan komputasi jumlah *bridge* pada graf dinamis inkremental.
3. Bagaimana hasil kinerja dari implementasi algoritma dan struktur data yang dirancang dalam penyelesaian permasalahan komputasi jumlah *bridge* pada graf dinamis inkremental.

1.3 Batasan Masalah

Permasalahan yang dibahas pada Tugas Akhir ini memiliki beberapa batasan, yaitu sebagai berikut:

1. Batas maksimum jumlah *vertex* awal pada graf adalah 50.000 *vertex*.
2. Batas maksimum jumlah operasi penambahan *edge* adalah 100.000 operasi.
3. *Edge* yang ditambahkan pada graf dipastikan bukan *edge* yang telah ada sebelumnya ataupun *edge* yang menghubungkan antara sebuah *vertex* dengan *vertex* itu sendiri.

1.4 Tujuan

Tujuan dari Tugas Akhir ini adalah sebagai berikut:

1. Melakukan analisis dan desain algoritma dan struktur data untuk menyelesaikan permasalahan komputasi jumlah *bridge* pada graf dinamis inkremental.
2. Mengevaluasi hasil dan kinerja dari algoritma dan struktur data yang telah dirancang dalam penyelesaian komputasi jumlah *bridge* pada graf dinamis inkremental.

1.5 Metodologi

Metodologi yang digunakan dalam pengerjaan Tugas Akhir ini adalah sebagai berikut:

1. Penyusunan proposal Tugas Akhir
   Pada tahap ini dilakukan penyusunan proposal tugas akhir yang berisi permasalahan dan gagasan solusi yang akan diteliti pada permasalahan komputasi jumlah *bridge* pada graf dinamis inkremental.
2. Studi literatur
   Pada tahap ini dilakukan pencarian informasi dan studi literatur mengenai pengetahuan atau metode yang dapat digunakan dalam penyelesaian masalah. Informasi didapatkan dari materi-materi yang berhubungan dengan algoritma dan struktur data yang digunakan untuk penyelesaian permasalahan ini, materi-materi tersebut didapatkan dari buku maupun internet.
3. Desain
   Pada tahap ini dilakukan desain rancangan algoritma dan struktur data yang digunakan dalam solusi untuk pemecahan masalah komputasi jumlah *bridge* graf dinamis inkremental.
4. Implementasi perangkat lunak
   Pada tahap ini dilakukan implementasi atau realiasi dari rancangan desain algoritma dan struktur data yang telah dibangun pada tahap desain ke dalam bentuk program.
5. Uji coba dan evaluasi

Pada tahap ini dilakukan uji coba kebenaran dan uji coba kinerja dari implementasi yang telah dilakukan. Pengujian kebenaran dilakukan pada sistem penilaian daring SPOJ sesuai dengan masalah yang dikerjakan untuk diuji apakah luaran dari program telah sesuai dengan luaran yang seharusnya. Pengujian kinerja dilakukan dengan cara memberi masukan pada implementasi program dengan variasi jumlah *vertex* dan operasi penambahan *edge* yang beragam untuk melihat pengaruh masing-masing jumlah *vertex* dan operasi terhadap pertumbuhan waktu eksekusi. Hasil dari uji coba kebenaran dan kinerja pada program digunakan sebagai bahan evaluasi kesalahan dan juga optimasi kinerja agar performa yang didapat lebih optimal.

6. Penyusunan buku Tugas Akhir
   Pada tahap ini dilakukan penyusunan buku Tugas Akhir yang berisi dokumentasi hasil pengerjaan Tugas Akhir.

1.6 Sistematika Penulisan

Berikut adalah sistematika penulisan buku Tugas Akhir ini:

1. BABI: PENDAHULUAN
   Bab ini berisi latar belakang, rumusan masalah, batasan masalah, tujuan, metodologi dan sistematika penulisan Tugas Akhir.
2. BAB II: DASAR TEORI
   Bab ini berisi dasar teori mengenai permasalahan dan algoritma penyelesaian yang digunakan dalam Tugas Akhir
3. BAB III: DESAIN
   Bab ini berisi desain algoritma dan struktur data yang digunakan dalam penyelesaian permasalahan.
4. BAB IV: IMPLEMENTASI
   Bab ini berisi implementasi berdasarkan desain algortima dan struktur data yang telah dilakukan pada tahap desain.

5. BAB V: UJI COBA DAN EVALUASI
   Bab ini berisi uji coba dan evaluasi dari hasil implementasi yang telah dilakukan pada tahap implementasi.
6. BAB VI: KESIMPULAN
   Bab ini berisi kesimpulan yang didapat dari hasil uji coba yang telah dilakukan.

BAB 2

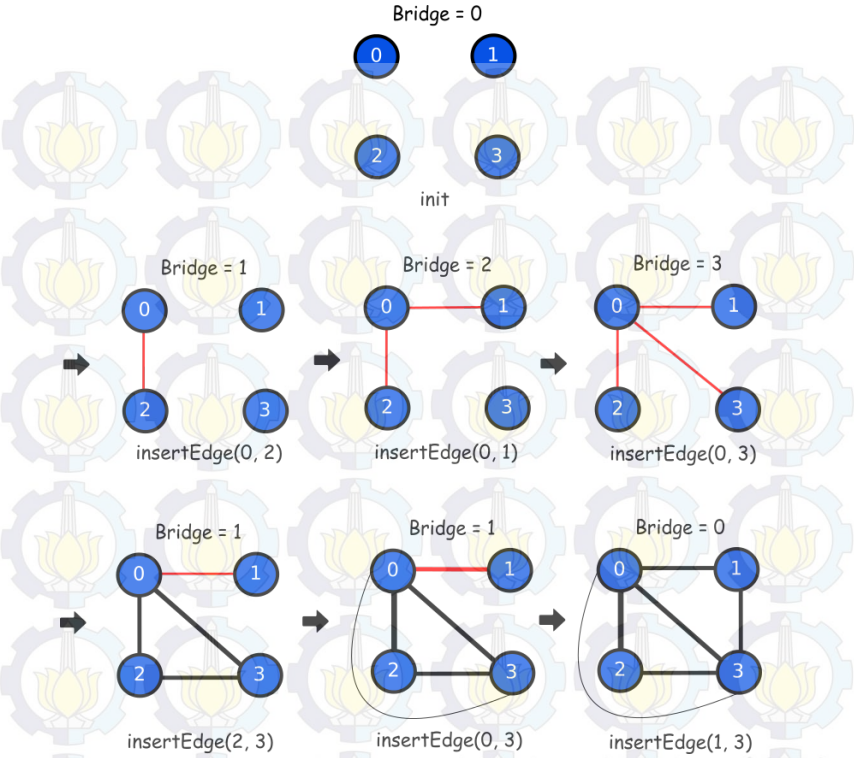
DASAR TEORI

Pada bab ini akan dijelaskan mengenai dasar teori yang menjadi dasar pengerjaan Tugas Akhir ini.

2.1 Deskripsi Permasalahan

Permasalahan yang diangkat pada Tugas Akhir ini adalah permasalahan komputasi jumlah *bridge* pada graf yang berubah secara dinamis karena operasi penambahan *edge* atau disebut juga graf dinamis inkremental. Diberikan sebuah *undirected graph* yang terdiri dari N *vertex* dan pada awalnya graf tersebut tidak memiliki *edge*. Selanjutnya akan dilakukan M operasi penambahan *undirected edge* yang menghubungkan antara sebuah *vertex* dengan *vertex* lainnya. Untuk setiap operasi penambahan *edge* diperlukan informasi total jumlah *bridge* yang terdapat pada graf tersebut. *Edge* yang ditambahkan untuk setiap operasi dipastikan bukan merupakan *edge* yang telah ada sebelumnya ataupun *edge* yang menghubungkan sebuah *vertex* dengan *vertex* itu sendiri.

Pada Gambar 2.1 diilustrasikan permasalahan di atas dengan jumlah *vertex* N bernilai 4 dan setiap *vertex* diberi nomor dari 0 hingga 3 dan akan dilakukan operasi penambahan *edge* dengan jumlah operasi M bernilai 6. Pertama dilakukan operasi penambahan *edge*(0, 2) yang menghubungkan antara *vertex* 0 dengan *vertex* 2, operasi ini akan membentuk *bridge* baru karena *edge* menghubungkan dua buah *connected component* yang berbeda sehingga jumlah *bridge* setelah operasi tersebut adalah 1. Selanjutnya dilakukan operasi penambahan *edge*(0, 1) dan juga *edge* (0, 3). Kedua operasi ini juga



Gambar 2.1: Ilustrasi permasalahan.

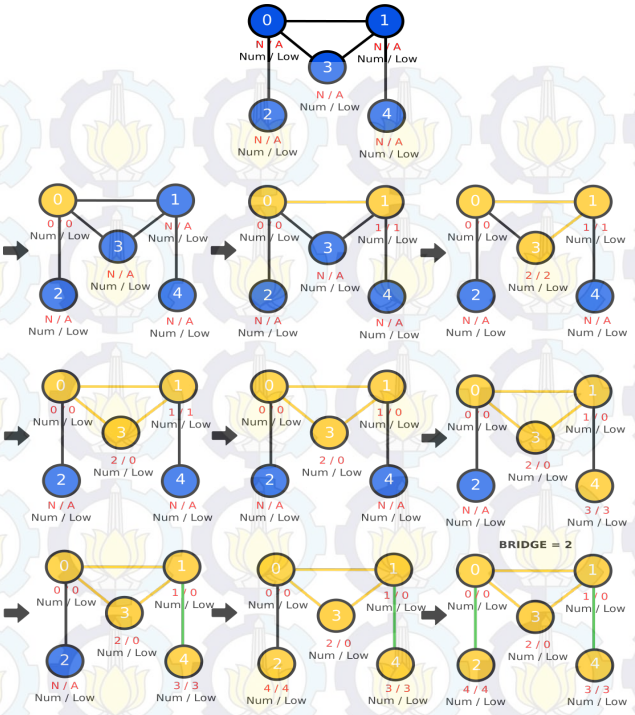
masing-masing membentuk sebuah *bridge* sehingga jumlah *bridge* pada dua operasi tersebut adalah 2 kemudian 3. Kemudian dilakuan operasi penambahan *edge*(2, 3). Operasi ini membentuk *cycle* antara *vertex* 0, 2 dan 3 sehingga *edge* yang ada pada *cycle* tersebut bukan lagi sebuah *bridge*, jumlah *bridge* setelah operasi ini adalah 1. Selanjutnya dilakukan operasi penambahan *edge*(0, 3) yang bukan merupakan sebuah *bridge* karena sudah terdapat *cycle* sebelumnya antara kedua *vertex* tersebut. Operasi selanjutnya adalah penambahan *edge*(1, 3) yang menciptakan *cycle* pada graf dan membuat

edge pada *cycle* tersebut bukan lagi sebuah *bridge*, jumlah *bridge* di akhir ilustrasi ini adalah 0. *Edge* yang berwarna merah pada Gambar 2.1 menandakan *edge* tersebut merupakan sebuah *bridge*, sedangkan *edge* berwarna hitam berarti *vertex*-*vertex* yang terhubung tergabung dalam sebuah *cycle*.

2.2 Strategi Penyelesaian Permasalahan

Solusi naif dari permasalahan ini adalah dilakukan komputasi ulang jumlah *bridge* untuk setiap operasi penambahan *edge* dengan cara mencoba menghapus setiap *edge* yang ada lalu memeriksa apakah graf tersebut masih terhubung dengan cara melakukan graf traversal. Jika graf tidak terhubung atau jumlah *connected component* bertambah dari pada bentuk graf sebelumnya sebelum *edge* dihapus maka *edge* tersebut merupakan sebuah *bridge*. Percobaan penghapusan *edge* ini dilakukan untuk setiap *edge* pada graf dan diulangi untuk setiap operasi pertambahan *edge*. Performansi metode naif untuk permasalahan ini membutuhkan waktu eksekusi yang kurang efisien dengan kompleksitas $\mathcal{O}(M^2(N+M))$.

Alternatif lain yang dapat dilakukan adalah dengan menggunakan algoritma *Bridge-finding* [2] yang sudah cukup terkenal untuk setiap operasi yang dilakukan dengan cara melakukan *depth-first-search* (DFS) traversal pada graf. Pada saat dilakukan DFS traversal *edge*(u, v) (dimisalkan u adalah *parent* dari v) dikatakan *bridge* apabila tidak ada *path* alternatif lain untuk mencapai u atau ancestor dari u dari subtree dengan v sebagai *root*. Ilustrasi dari komputasi jumlah *bridge* pada suatu graf menggunakan algoritma *Bridge-Finding* diilustrasikan pada Gambar 2.2. Algoritma ini dijalankan untuk setiap operasi penambahan *edge* untuk mengetahui jumlah *bridge* pada setiap operasi. Pendekatan metode ini untuk menyelesaikan permasalahan di atas masih memiliki kompleksitas $\mathcal{O}(M(N+M))$ yang masih belum cukup efisien.



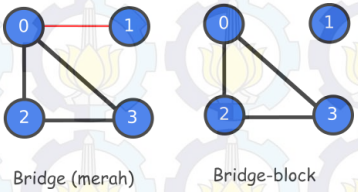
Gambar 2.2: Ilustrasi komputasi jumlah *bridge* menggunakan algoritma *Bridge-Finding*.

Pada kasus komputasi jumlah *bridge* pada graf dinamis inkremental, solusi yang lebih efisien bisa didapatkan dengan memperhatikan beberapa kondisi ketika terjadi operasi penambahan *edge* [3].

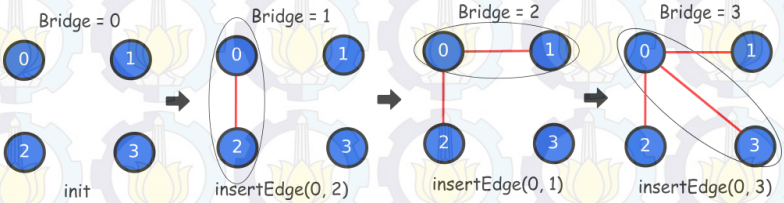
Dimisalkan *bridge-block* adalah *connected component* yang terbentuk apabila seluruh *bridge* pada graf dihapus seperti yang dijelaskan pada Gambar 2.3, terdapat tiga kondisi yang harus diperhatikan untuk setiap operasi pendambahan *edge*(u, v) yaitu:

1. *Vertex* u dan v berada pada himpunan *connected component* yang berbeda.

2. *Vertex* u dan v berada pada himpunan *bridge-block* yang berbeda namun tergabung dalam himpunan *connected component* yang sama.
3. *Vertex* u dan v berada pada himpunan *bridge-block* yang sama.



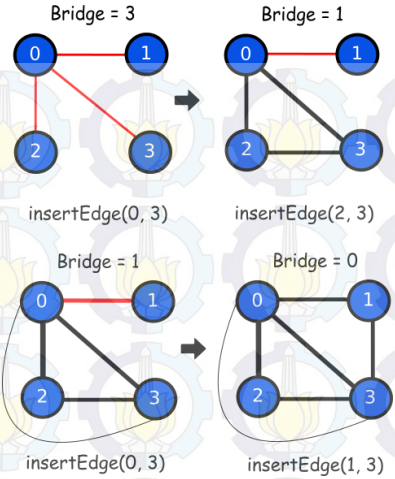
Gambar 2.3: Ilustrasi *bridge*, dan *bridge-block*.



Gambar 2.4: Ilustrasi penambahan *edge*(u, v) dengan u dan v berada pada *connected component* yang berbeda.

Jika u dan v berada pada himpunan *connected component* yang berbeda seperti yang diilustrasikan pada Gambar 2.4, maka *edge* yang ditambahkan adalah *bridge* dan *vertex* u dan v akan tergabung dalam himpunan *connected component* yang sama tetapi masih dalam himpunan *bridge-block* yang berbeda karena belum terdapat *cycle* antara u dan v.

Jika *vertex* u dan v berada pada himpunan *bridge-block* yang berbeda namun tergabung dalam himpunan *connected component* yang sama, maka *edge* tersebut akan membentuk *cycle* baru yang akan mengurangi jumlah *bridge* pada graf. Semua *bridge-block* yang



Gambar 2.5: Ilustrasi penambahan *edge*(u, v) dengan u dan v berada pada *bridge-block* yang berbeda namun dalam *connected component* yang sama.

berada di antara *path*(u, v) akan tergabung menjadi satu *bridge-block* yang sama dan semua *bridge* yang terdapat di antara *path*(u, v) bukan lagi sebuah *bridge*. Kondisi ini diilustrasikan pada Gambar 2.5.

Jika *vertex* u dan v berada pada himpunan *bridge-block* yang sama maka tidak ada perubahan yang terjadi pada struktur data begitu juga dengan jumlah *bridge* pada graf karena sudah ada *cycle* yang terjadi sebelumnya antara *path*(u, v).

Dari pendekatan metode penyelesaian di atas, operasi-operasi yang akan dilakukan adalah:

1. Menentukan apakah dua buah *vertex* terdapat dalam himpunan *connected component* yang sama atau tidak.
2. Menentukan apakah dua buah *vertex* terdapat dalam himpunan *bridge-block* yang sama atau tidak.

3. Mendapatkan *path* antara *bridge-block* yang saling terhubung.
4. Menggabungkan dua himpunan *connected component* menjadi satu himpunan.
5. Menggabungkan beberapa himpunan *bridge-block* menjadi satu himpunan.

Untuk mendukung operasi di atas bisa digunakan struktur data dengan bantuan *disjoint set* yang menyimpan himpunan *bridge-block* dan *connected component* dari suatu *vertex* dan penerapan Lowest Common Ancestor untuk mendapatkan *path* antara *bridge-block* yang saling terhubung dalam sebuah *connected component*. Dengan pendekatan heuristik *union by weight* dan *path-compression* pada *disjoint set*, pendekatan penyelesaian di atas memiliki kompleksitas waktu *amortized* $\mathcal{O}(M(\alpha(N)))$ dimana $\alpha(N)$ adalah invers dari *fast-growing ackermann function* dan bernilai ≤ 4 untuk setiap N [4]. Kompleksitas dari pendekatan penyelesaian ini sudah memiliki ekspektasi waktu eksekusi yang cukup efisien.

2.3 *Disjoint Set*

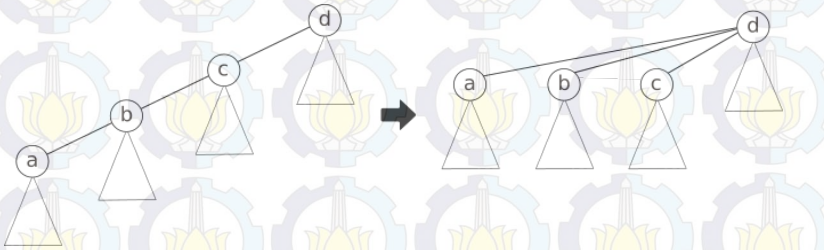
Disjoint set adalah struktur data yang merepresentasikan kumpulan dari himpunan objek yang saling lepas [2]. *Disjoint set* digunakan untuk mengetahui himpunan terkait dari suatu objek. Pada dasarnya *disjoint set* mendukung tiga operasi dasar, yaitu:

- *Make-Set*: Membuat sebuah objek baru yang tidak terkait dengan himpunan yang sudah ada sebelumnya.
- *Find-Set*: Menentukan himpunan terkait dari suatu objek.
- *Union*: Menggabungkan dua buah himpunan menjadi satu.

Penerapan *disjoint set* pada graf dilakukan dengan cara setiap *vertex* memiliki *parent* yang merujuk pada himpunan terkait dari *vertex* tersebut. Setiap himpunan pada *disjoint set* direpresentasikan oleh salah satu anggota dari himpunan tersebut untuk selanjutnya disebut

root. Himpunan dari suatu *vertex* diketahui dengan cara menelusuri *parent* pointer hingga mencapai *root*. Proses penggabungan dua buah himpunan menjadi satu dilakukan dengan cara menjadikan salah satu *vertex* pada himpunan menjadi *parent* dari *root* himpunan yang lain.

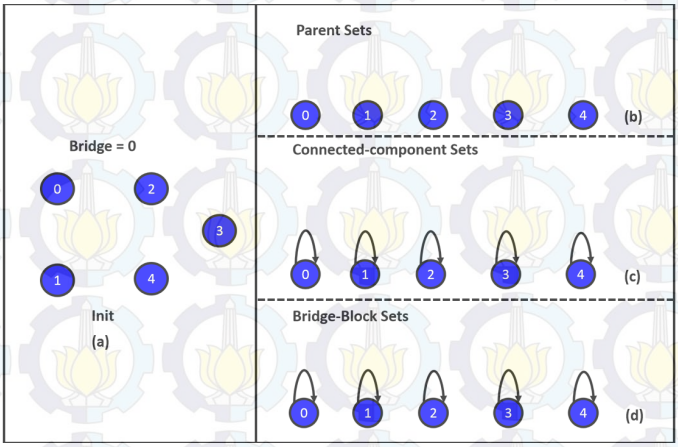
Terdapat dua pendekatan heuristik yang bisa diterapkan pada *disjoint set* untuk mendapatkan kinerja waktu yang lebih optimal. Pendekatan heuristik yang pertama adalah dengan melihat karakteristik kedua himpunan ketika akan dilakukan proses penggabungan. Salah satu metode pada pendekatan ini adalah *union by weight*. Pada *union by weight* penggabungan dua himpunan akan melihat ukuran dari kedua himpunan. *Root* dari himpunan dengan ukuran yang lebih kecil akan menjadi child dari *node* pada himpunan dengan ukuran yang lebih besar. Pendekatan heuristik yang kedua adalah *path-compression*. Seperti yang diilustrasikan pada Gambar 2.6 ketika proses *Find-set* terjadi maka setiap *vertex* yang ada dalam *path* dari *vertex* awal menuju *root* pada akhirnya akan merujuk pada *root* dari *tree*. Dengan menggunakan kedua pendekatan heuristik *union by weight* dan *path-compression* maka kompleksitas dari *disjoint-set* untuk setiap operasi *Find-Set* dan *Union* adalah *amortized* $\alpha(M, N)$ dimana $\alpha(M, N)$ adalah invers dari *fast-growing ackermann function* dan bernilai ≤ 4 untuk setiap N [4].



Gambar 2.6: *Path-compression* pada *disjoint set*.

Aplikasi dari *disjoint set* pada permasalahan ini adalah untuk me-

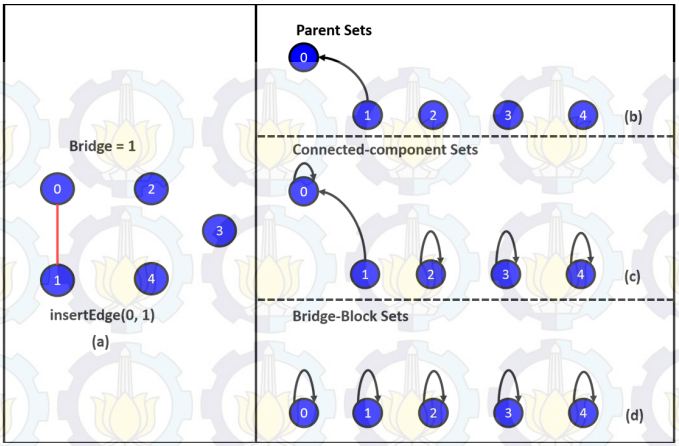
ngetahui himpunan *bridge-block* dan *connected component* dari sebuah *vertex*. Pada Gambar 2.7 diilustrasikan inisialisasi dari sebuah graf yang awalnya tidak memiliki *edge* dan representasi *disjoint set bridge-block* dan *connected component* dari graf.



Gambar 2.7: Ilustrasi *disjoint set* dari graf pada permasalahan.

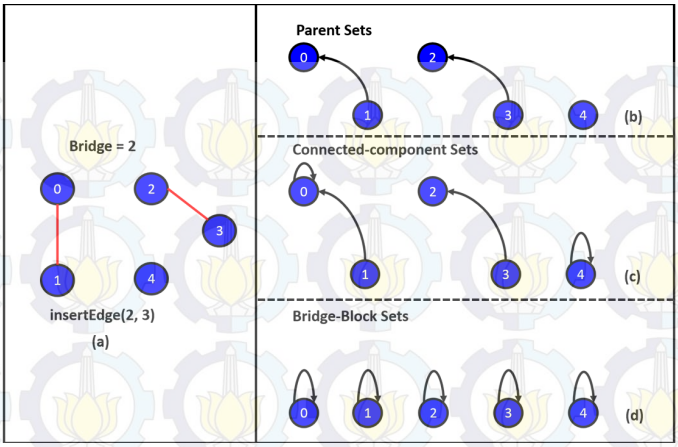
Parent sets pada ilustrasi merupakan *disjoint set* dari *connected component* tanpa *path-compression*. *Parent* sets dibutuhkan ketika proses penggabungan beberapa *bridge-block* menjadi satu akibat terbentuknya *cycle* dan mengakibatkan berkurangnya jumlah *bridge*. *Parent* sets menjaga struktur hubungan antara *bridge-block*.

Seperti yang telah dijelaskan pada subbab 2.2 ketika terjadi proses penambahan *edge* maka akan dilihat *bridge-block root* dari kedua *vertex*. Jika kedua *vertex* tidak dalam *bridge-block* yang sama maka akan dilihat *connected component root* dari kedua *vertex*. Jika kedua *vertex* tidak dalam *connected component* yang sama maka *edge* tersebut adalah *bridge* dan *vertex* dengan ukuran *connected component* yang lebih kecil akan menjadi *root* dari himpunan *connected*

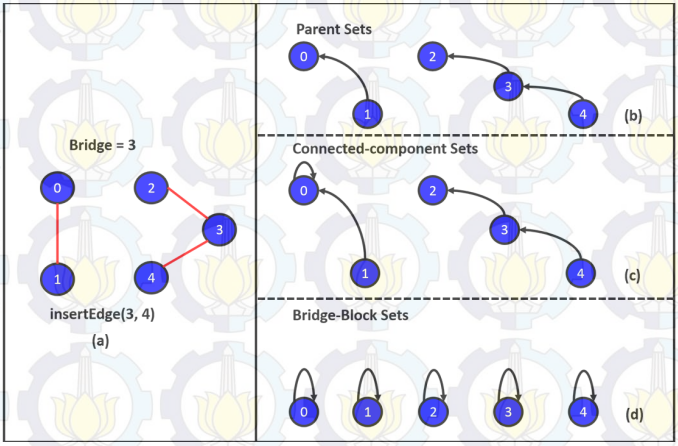


Gambar 2.8: Ilustrasi penambahan *bridge* dan representasi pada *disjoint set* (1).

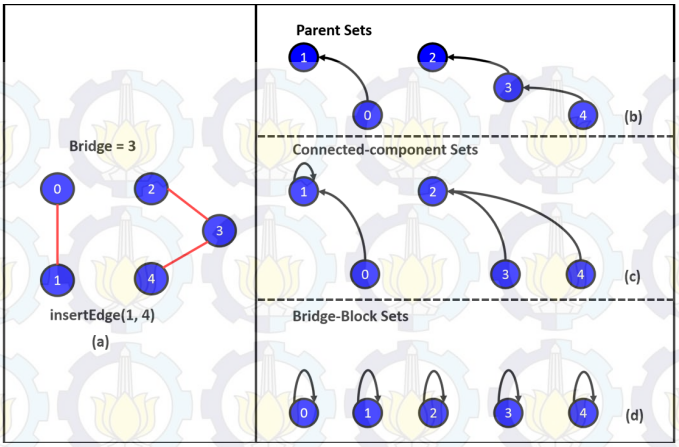
component. Untuk mengetahui *bridge-block* dan *connected component* terkait dari suatu *vertex* digunakan operasi dasar *Find-set* pada *disjoint set* dengan tambahan *path-compression* untuk *connected component* sets dan *bridge-block* sets namun tidak pada *parent* sets. Proses menjadikan *vertex* sebagai *root* dari himpunan *connected component* dilakukan dengan cara membalikkan arah pointer *parent* pada *parent* sets dan *connected component* sets dari semua *bridge-block root* yang ada dalam *path* dari *bridge-block root vertex* tersebut ke *root* dari *connected component*. Untuk mengetahui *bridge-block* selanjutnya dalam *path* menuju *root connected component* maka akan dilihat *parent* dari *vertex* tersebut pada *parent* sets. Setelah menjadi *root* dari himpunan *connected-component* maka *parent* pada *parent* sets dan *connected component* sets dari *vertex* tersebut adalah *bridge-block root* dari *vertex* dengan ukuran himpunan *connected-component* yang lebih besar. Ilustrasi dari kondisi di atas digambarkan juga pada Gambar 2.9, 2.10, 2.11 dan 2.12.



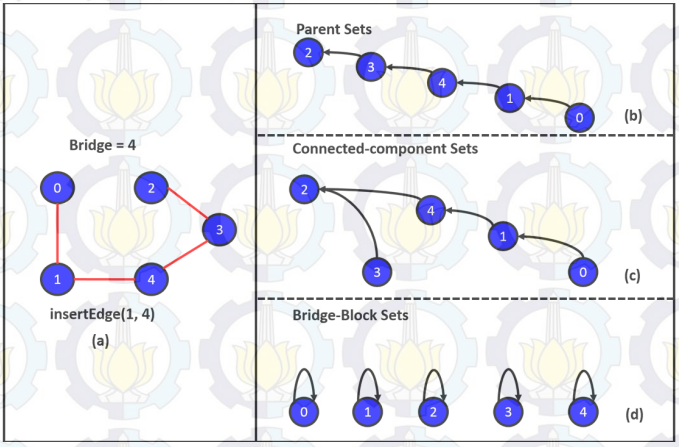
Gambar 2.9: Ilustrasi penambahan *bridge* dan representasi pada *disjoint set* (2).



Gambar 2.10: Ilustrasi penambahan *bridge* dan representasi pada *disjoint set* (3).



Gambar 2.11: Ilustrasi penambahan *bridge* dan representasi pada *disjoint set* (4).



Gambar 2.12: Ilustrasi penambahan *bridge* dan representasi pada *disjoint set* (5).

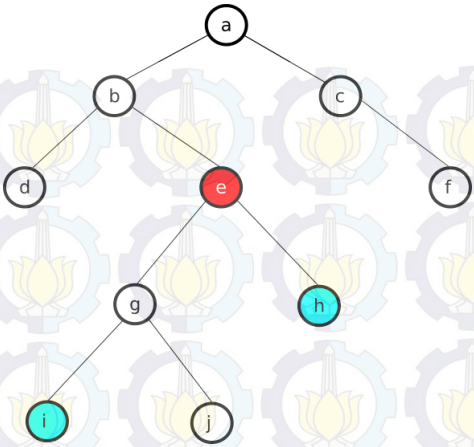
Pada saat penambahan *edge*(1, 4) seperti yang ditunjukkan pada

Gambar 2.11 dan 2.12. *Vertex* 1 memiliki ukuran himpunan *connected component* yang lebih kecil dibandingkan dengan ukuran himpunan *connected component* dari *vertex* 4. Ukuran himpunan dapat diketahui dengan melihat nilai ukuran subtree dari *disjoint set* yang disimpan pada *root* dari *set*. Pada kondisi ini juga dilakukan *path-compression* pada *disjoint set* sehingga *vertex* 4 langsung merujuk pada *vertex* 2 yang merupakan *root* dari himpunan *connected component* sedangkan *vertex* 1 sebelumnya sudah langsung merujuk pada *root* dari *connected component* yaitu *vertex* 0. *Vertex* 1 yang memiliki ukuran himpunan *connected component* lebih kecil terlebih dahulu akan menjadi *root* dari himpunan *connected component*. Proses ini dilakukan dengan cara membalikkan arah pointer *parent* pada *parent* sets dan *connected component* sets dari semua *bridge-block root* yang ada dalam *path* dari *bridge-block root vertex* tersebut ke *root* dari *connected component* (Gambar 2.11). Setelah *vertex* 1 telah menjadi *root* dari himpunan *connected component* maka *root bridge-block* dari *vertex* dengan ukuran himpunan *connected component* yang lebih besar yaitu *vertex* 4 akan menjadi *parent* dari *vertex* 1 pada *connected component* sets dan *parent* sets (Gambar 2.12).

2.4 Lowest Common Ancestor (LCA)

LCA adalah sebuah konsep dalam teori graf dan ilmu komputer. LCA antara dua *vertex* u dan v adalah *vertex* terjauh dari *tree* T yang memiliki u dan v sebagai keturunan [5]. LCA dalam aplikasinya dapat digunakan untuk mencapai titik temu terdekat antara dua buah *vertex* dalam sebuah *tree*. Pada Gambar 2.13 diilustrasikan bahwa *vertex* e adalah LCA antara *vertex* h dan i dalam sebuah *tree*.

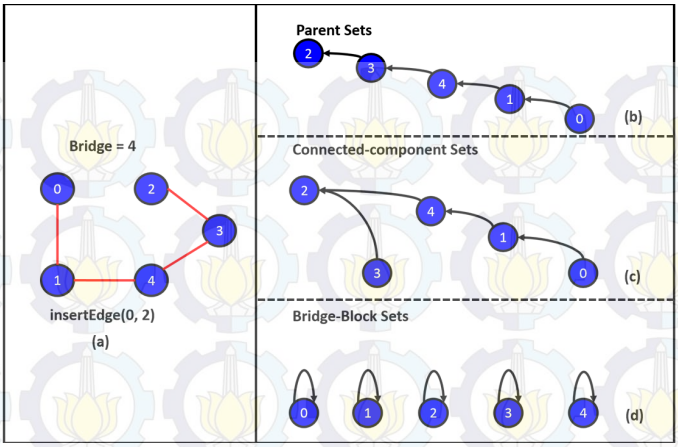
Aplikasi dari LCA pada permasalahan ini adalah ketika proses penggabungan beberapa *bridge-block* menjadi satu akibat dari terbentuknya *cycle* ketika penambahan *edge*. Semua *bridge-block* yang berada di antara *path* kedua *vertex* akan tergabung menjadi sa-



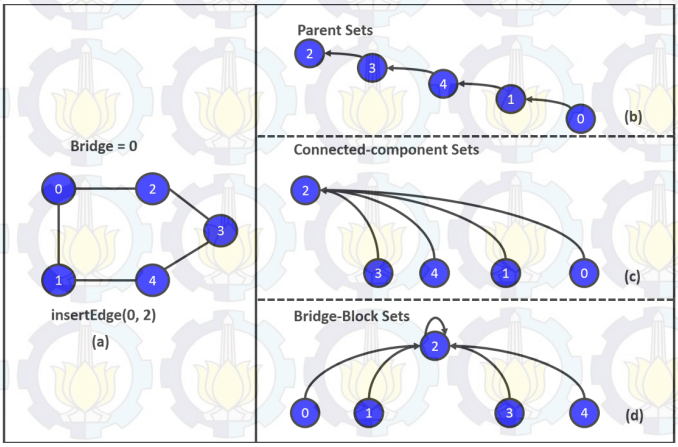
Gambar 2.13: Contoh ilustrasi LCA dari 2 *vertex*.

tu *bridge-block* dan semua *bridge* yang terdapat di antara *path* tersebut bukan lagi sebuah *bridge*. Untuk mengetahui *bridge-block* yang ada dalam *path* antara kedua *vertex* maka dicari LCA antara kedua *vertex*. Pencarian LCA pada kasus ini dilakukan dengan cara kedua *vertex* mencari *bridge-block root* lalu melihat ke arah *bridge-block* selanjutnya dengan cara melihat *parent* dari *vertex* tersebut pada parents *set*. Kemudian dari *parent* tersebut mencari lagi *bridge-block root* dan proses ini berjalan terus menerus hingga mencapai titik temu antara kedua *vertex*. Titik temu ini adalah LCA dari kedua *vertex* dan *parent bridge-block* dari semua *bridge-block* yang telah dilewati sebelumnya akan merujuk pada LCA.

Pada Gambar 2.14 dan 2.15 diilustrasikan operasi penambahan *edge*(0, 2) yang akan membentuk *cycle*. LCA antara *bridge-block vertex* 0 dan 2 adalah *vertex* 2. Semua *bridge-block parent* yang ada dalam *path* ini yaitu *vertex* 0, 1, 4, 3 dan 2 akan merujuk pada LCA yaitu *vertex* 2 (Gambar 2.15).



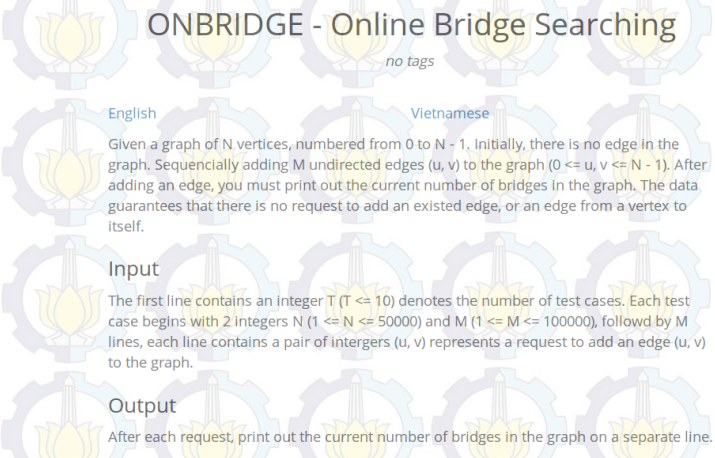
Gambar 2.14: Ilustrasi *cycle* dan penerapan LCA pada permasalahan (1).



Gambar 2.15: Ilustrasi *cycle* dan penerapan LCA pada permasalahan (2).

2.5 Permasalahan *Online Bridge Searching* pada SPOJ

Pada situs penilaian daring SPOJ terdapat permasalahan komputasi jumlah *bridge* pada graf secara *online* dengan judul soal *Online Bridge Searching* dan kode soal ONBRIDGE [6]. Deskripsi permasalahan ditunjukkan pada Gambar 2.16.



ONBRIDGE - Online Bridge Searching

no tags

English Vietnamese

Given a graph of N vertices, numbered from 0 to N - 1. Initially, there is no edge in the graph. Sequencially adding M undirected edges (u, v) to the graph (0 <= u, v <= N - 1). After adding an edge, you must print out the current number of bridges in the graph. The data guarantees that there is no request to add an existed edge, or an edge from a vertex to itself.

Input

The first line contains an integer T (T <= 10) denotes the number of test cases. Each test case begins with 2 integers N (1 <= N <= 50000) and M (1 <= M <= 100000), followd by M lines, each line contains a pair of intergers (u, v) represents a request to add an edge (u, v) to the graph.

Output

After each request, print out the current number of bridges in the graph on a separate line.

Gambar 2.16: Deskripsi permasalahan *Online Bridge Searching* di SPOJ.

Deskripsi singkat dari persoalan tersebut ialah diberikan sebuah *undirected graph* yang terdiri dari N buah *vertex* dan pada awalnya graf tersebut tidak memiliki *edge*. Pada graf tersebut akan dilakukan M operasi penambahan *undirected edge* yang menghubungkan antara sebuah *vertex* dengan *vertex* lainnya. Untuk setiap operasi penambahan *edge* dibutuhkan informasi total jumlah *bridge* yang terdapat pada graf tersebut. *Edge* yang ditambahkan dipastikan bukan merupakan *edge* yang telah ada sebelumnya ataupun *edge* yang menghubungkan sebuah *vertex* dengan *vertex* itu sendiri.

Berikut merupakan format masukan yang diminta dari permasalahan:

1. Baris pertama terdiri dari sebuah bilangan integer T yang merepresentasikan banyaknya kasus uji
2. Untuk setiap kasus uji terdapat dua buah integer N dan M yang menyatakan banyaknya *vertex* pada graf dan jumlah operasi penambahan *edge* yang akan dilakukan.
3. M baris selanjutnya adalah berisi pasangan U dan V yang merepresentasikan dua buah *vertex* yang mengalami penambahan *edge*.

Format keluaran dari permasalahan tersebut adalah sebuah integer yang merepresentasikan jumlah *bridge* pada graf untuk setiap operasi penambahan *edge*.

Batasan permasalahan yang diberikan adalah sebagai berikut:

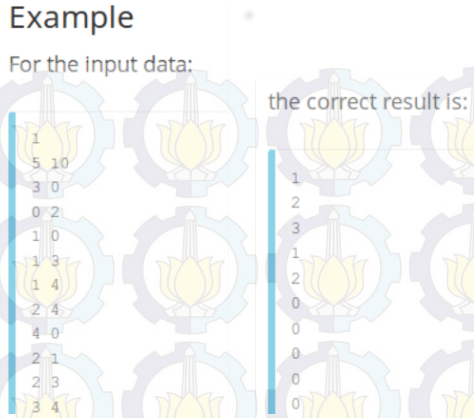
1. $T \leq 10$
2. $1 \leq N \leq 50000$
3. $1 \leq M \leq 100000$
4. $0 \leq u, v \leq N - 1$

Program akan diuji pada *cluster Cube* (Intel Pentium G860 3GHz) dengan batasan waktu eksekusi 0.140s, batasan kapasitas *memory* sebesar 1536MB dan batasan kode sumber sebesar 50000B.

Pada deskripsi soal tersebut dijelaskan juga contoh masukan dan keluaran yang akan digunakan sesuai dengan format masukan dan keluaran yang telah dijelaskan. Contoh masukan dan keluaran yang diberikan digambarkan pada Gambar 2.17.

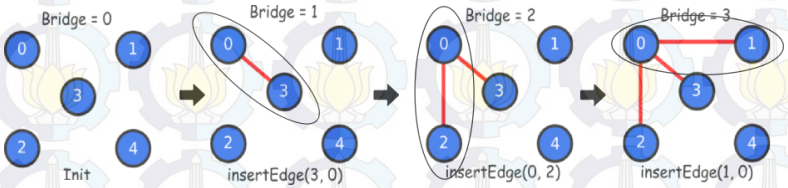
Untuk menyelesaikan permasalahan pada contoh maka akan dibangun struktur data dan melihat kondisi *vertex* yang mengalami pertambahan *edge* seperti yang telah dijelaskan pada subbab 2.2.

Pada contoh masukan pertama dilakukan operasi penambahan *edge*(3, 0) yang menghubungkan antara *vertex* 3 dengan *vertex* 0, operasi ini akan membentuk *bridge* baru karena *edge* menghu-



Gambar 2.17: Contoh masukan dan keluaran pada soal *Online Bridge Searching*.

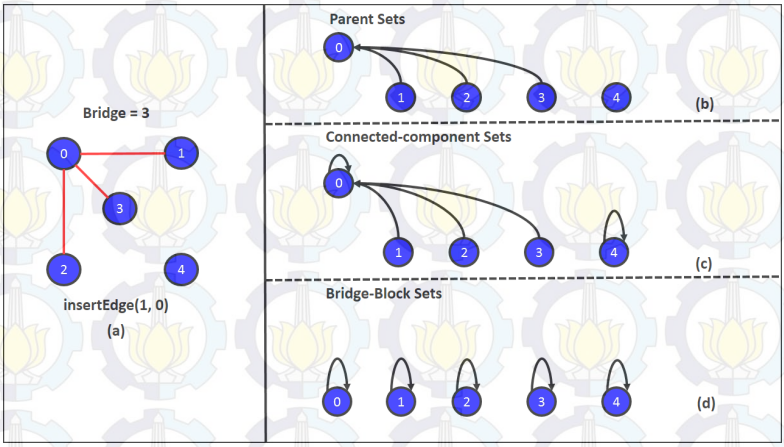
bungkan dua buah *connected component* yang berbeda sehingga jumlah *bridge* setelah operasi tersebut adalah 1. Selanjutnya dilakukan operasi penambahan *edge*(0, 2) dan juga *edge* (1, 0). Kedua operasi ini juga masing-masing memiliki kondisi yang sama dengan kondisi sebelumnya dengan membentuk sebuah *bridge* sehingga jumlah *bridge* daripada dua urutan operasi tersebut adalah 2 kemudian 3. Ketiga operasi penambahan *bridge* ini diilustrasikan pada Gambar 2.18.



Gambar 2.18: Ilustrasi operasi yang menghasilkan *bridge* pada contoh permasalahan.

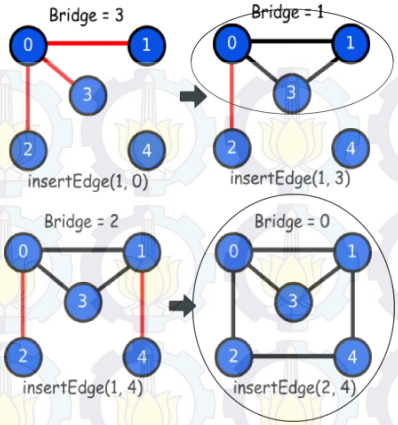
Ilustrasi *disjoint set* dari kondisi graf setelah dilakukan ketiga ope-

rasi di atas digambarkan pada Gambar 2.19.



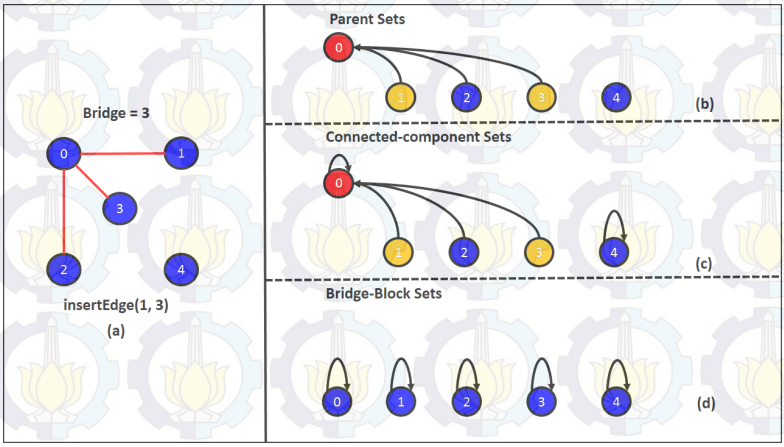
Gambar 2.19: Ilustrasi *disjoint set* pada contoh permasalahan.

Kemudian dilakuan operasi penambahan *edge*(1, 3). Operasi ini menghubungkan dua buah *vertex* yang berada pada *bridge-block* yang berbeda namun terhimpun dalam *connected component* yang sama sehingga membentuk *cycle* antara *vertex* 0, 1 dan 3. Operasi ini mengakibatkan *edge* yang ada dalam *path cycle* tersebut bukan lagi sebuah *bridge*. Untuk mendapatkan *path* antara *bridge-block* yang sebelumnya sudah berada dalam sebuah *connected component* yang sama maka dilakukan pencarian LCA *bridge-block* dari masing-masing *vertex* seperti yang telah dijelaskan pada subbab 2.2. Pada awalnya kedua *vertex* akan melihat pada *bridge-block root* masing-amsing lalu untuk mendapatkan *vertex* selanjutnya yang harus dikunjungi akan dilihat pada *parent* sets dari *vertex* tersebut. Proses ini terus dilakukan hingga mencapai titik temu antara kedua *vertex*. Setelah didapatkan LCA maka *bridge-block* yang telah dikunjungi dalam proses ini pada akhirnya akan merujuk pada LCA dan jumlah *bridge* akan berkurang sebanyak jumlah *bridge-block* yang mengarah ke LCA kecuali LCA itu sendiri. Jumlah *bridge* se-

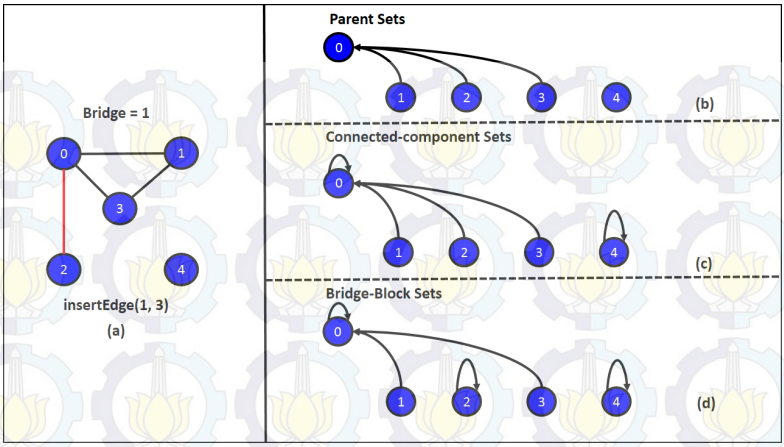


Gambar 2.20: Ilustrasi operasi yang menggabungkan beberapa *bridge-block* dan menghapus *bridge* pada contoh permasalahan.

telah operasi ini adalah 1. Ilustrasi struktur *disjoint set* pada operasi di atas digambarkan pada Gambar 2.21 dan 2.22.



Gambar 2.21: Ilustrasi *disjoint set* dan LCA saat pembentukan *cycle* (1).

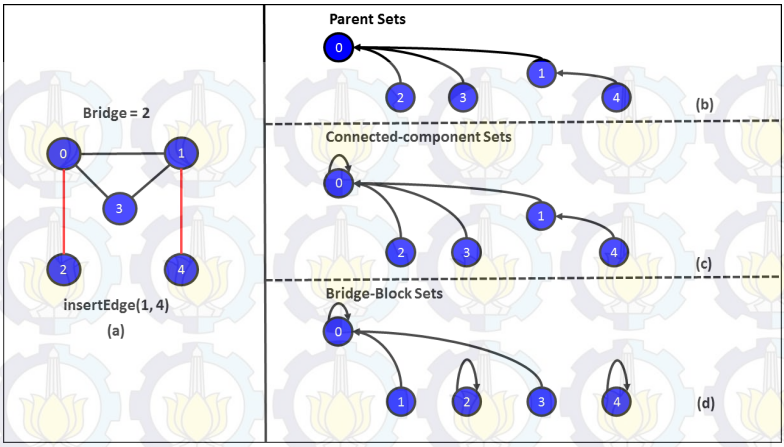


Gambar 2.22: Ilustrasi *disjoint set* dan LCA saat pembentukan *cycle* (2).

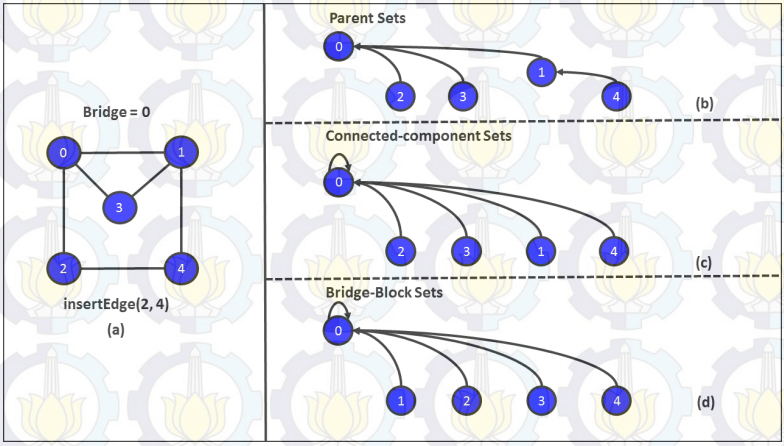
Operasi penambahan *edge*(1, 4) membentuk sebuah *bridge* karena *vertex* 1 dan 4 berada pada himpunan *connected component* yang berbeda. Karena *vertex* 4 memiliki ukuran himpunan *connected component* yang lebih kecil daripada *vertex* 1 maka *vertex* 4 akan merujuk pada *bridge-block root* dari *vertex* 1 yaitu *vertex* 0. Total *bridge* setelah operasi ini adalah 2. Ilustrasi dari operasi di atas digambarkan pada Gambar 2.23.

Operasi penambahan *edge*(2, 4) membentuk *cycle* pada graf yang membuat *edge* pada *cycle* tersebut bukan lagi sebuah *bridge*, sehingga jumlah *bridge* setelah operasi ini adalah 0. Ilustrasi struktur data *disjoint set* setelah operasi penambahan *edge*(2, 4) digambarkan pada Gambar 2.24.

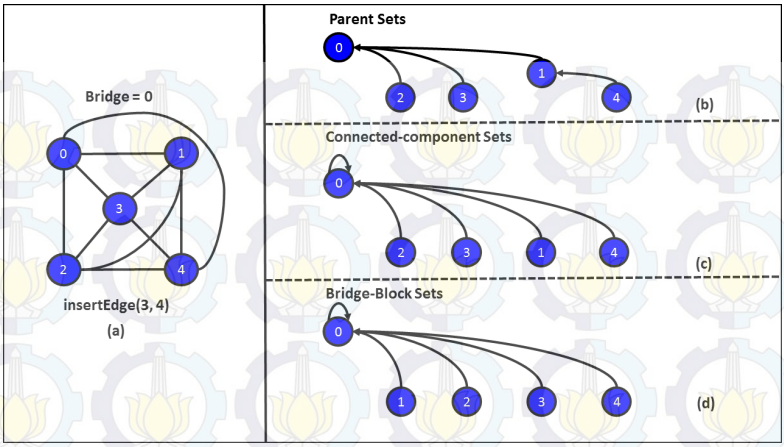
Operasi penambahan *edge*(4, 0), *edge*(2, 1), *edge*(2,3), dan *edge*(3, 4) menghubungkan *vertex* yang sudah tergabung dalam *bridge-block* yang sama sehingga tidak mempengaruhi jumlah *bridge* pada graf. Kondisi akhir dari graf dan struktur data pada akhir operasi digambarkan pada Gambar 2.25.



Gambar 2.23: Ilustrasi graf dan *disjoint set* setelah operasi penambahan *edge*(1, 4).



Gambar 2.24: Ilustrasi graf dan *disjoint set* setelah operasi penambahan *edge*(2, 4).



Gambar 2.25: Ilustrasi kondisi akhir dari graf dan *disjoint set* pada contoh kasus *Online Bridge Searching*.

2.6 Pembuatan Data Generator untuk Uji Coba

Pembuatan data generator dilakukan untuk membuat kasus uji yang akan digunakan untuk pengujian pada uji coba yang akan dijelaskan pada bab 5. Data generator ini disesuaikan dengan format masukan yang sudah dijelaskan pada subbab 2.5.

Terdapat 2 skenario yang akan dilakukan untuk uji coba, skenario tersebut ialah pengaruh jumlah *vertex* terhadap waktu dan pengaruh jumlah operasi pertambahan *edge* terhadap waktu. Untuk skenario pertama akan dibuat data generator dengan banyak jumlah operasi tetap dan jumlah *vertex* dari 1.000 hingga 50.000 dengan rentang 1.000. Pseudocode dari data generator untuk skenario pertama dapat dilihat pada Gambar 2.26.

Untuk skenario kedua akan dibuat data generator dengan banyak jumlah *vertex* tetap dan jumlah operasi dari 2.000 hingga 100.000 dengan rentang 2.000. Pseudocode dari data generator untuk skenario kedua dapat dilihat pada Gambar 2.27.

```
1   let used[0..MAXN][0..MAXN] be a new array
2   A = 50
3   for a = 1 to A
4       T = 10
5       for i = 0 to MAXN
6           for j = 0 to MAXN
7               used[i][j] = false
8       N = a * 1000
9       M = 100000 // constant M value
10      for t = 1 to T
11          writeToFile(N)
12          writeToFile(M)
13          for i = 1 to M
14              u = rand()
15              v = rand()
16              while used[u][v] is true
17                  u = rand()
18                  v = rand()
19              writeToFile(u)
20              writeToFile(v)
```

Gambar 2.26: Pseudocode Fungsi Main Generator Skenario Uji 1

```
1   let used[0..MAXN][0..MAXN] be a new array
2   A = 50
3   for a = 1 to A
4       T = 10
5       for i = 0 to MAXN
6           for j = 0 to MAXN
7               used[i][j] = false
8       N = 50000 // constant N value
9       M = a * 2000
10      for t = 1 to T
11          writeToFile(N)
12          writeToFile(M)
13          for i = 1 to M
14              u = rand()
15              v = rand()
16              while used[u][v] is true
17                  u = rand()
18                  v = rand()
19              writeToFile(u)
20              writeToFile(v)
```

Gambar 2.27: Pseudocode Fungsi Main Generator Skenario Uji 2

Halaman ini sengaja dikosongkan

BAB 3

DESAIN

Pada bab ini akan dijelaskan desain sistem yang digunakan untuk menyelesaikan permasalahan pada Tugas Akhir ini.

3.1 Deskripsi Umum Sistem

Sistem akan menerima masukan jumlah kasus uji. Untuk setiap kasus uji, sistem akan menerima masukan jumlah *vertex*, jumlah *edge* dan rangkaian pasangan *vertex* yang mengalami penambahan *edge*.

```
1   let node[0..MAXN] be a new array
2   T = Input() // Total testcase
3   for t = 1 to T
4       N = Input() // Total vertex in the graph
5       M = Input() // Total add edge operation
6       bridge = 0
7       for i = 0 to N − 1
8           Init(node[i])
9       for i = 0 to M − 1
10          u = Input()
11          v = Input()
12          InsertEdge(u, v) // Edge added between u and v
13          Print(bridge)
```

Gambar 3.1: Pseudocode Fungsi Main

Pada awalnya sistem akan menginisialisasi nilai dari kelas *node* yang merupakan representasi *vertex* pada graf. Untuk setiap masukan penambahan *edge* terhadap pasangan *vertex* sistem akan menjalankan fungsi InsertEdge(u, v) yang akan melakukan penambahan *edge* dan komputasi jumlah *bridge* lalu menampilkan luaran jumlah *bridge* yang terdapat pada graf. Pseudocode Fungsi Main ditunjukkan pada Gambar 3.1.

3.2 Desain Kelas *Node* dan Fungsi Init

Kelas *Node* merepresentasikan *vertex* pada graf. *Node* memiliki pointer yang akan merujuk pada *parent* dari *vertex*, pointer yang akan merujuk pada himpunan *bridge-block parent* dari *vertex* dan pointer yang akan merujuk pada himpunan *connected component* dari *vertex*. *Node* juga menyimpan nilai ukuran *connected component* yang tergabung dengan *vertex* tersebut dan nilai counter yang akan digunakan untuk penomoran *vertex* ketika proses penggabungan *bridge-block*. Pseudocode Fungsi Init ditunjukkan pada Gambar 3.2

```
1  u.parent = NULL
2  u.bParent = u
3  u.cParent = u
4  u.cSize = 1
5  u.counter = 0
```

Gambar 3.2: Pseudocode Fungsi Init

Fungsi Init akan mengatur nilai inisialisasi *node*. Pada tahap inisialisasi awal, nilai pointer *parent* akan merujuk pada NULL. Pointer *bridge-block* dan *connected component* akan merujuk pada *node* itu sendiri yang menandakan bahwa *vertex* tersebut merupakan *root* dari himpunan *bridge-block* dan *connected component node* tersebut.

Nilai ukuran *connected component* bernilai satu dan nilai counter untuk inisialisasi bernilai nol.

3.3 Desain Fungsi InsertEdge

Fungsi InsertEdge menerima dua parameter u dan v yaitu *vertex* yang mengalami pertambahan *edge*. Fungsi InsertEdge akan melihat kondisi pasangan *vertex* u dan v sebelum mengalami pertambahan *edge* sesuai dengan kondisi yang telah dijelaskan pada subbab 2.2 . Pseudocode dari Fungsi InsertEdge dijelaskan pada Gambar 3.3.

```
1   bU = FindBridgeBlock(u)
2   bV = FindBridgeBlock(v)
3   if bU == bV
4       return
5   else cU = FindComponent(u)
6       cV = FindComponent(v)
7       if cU == cV
8           MergeBridgeBlock(bU, bV)
9       else bridge = bridge + 1
10          if cU.cSize < cV.cSize
11              SetRoot(bU)
12              bU.parent = bV
13              bU.cParent = bV
14              cV.cSize = cV.cSize + cU.cSize
15          else SetRoot(bV)
16              bV.parent = bU
17              bV.cParent = bU
18              cU.cSize = cU.cSize + cV.cSize
```

Gambar 3.3: Pseudocode Fungsi InsertEdge

3.4 Desain Fungsi FindBridgeBlock

Fungsi FindBridgeBlock akan memberikan nilai kembali berupa himpunan *bridge-block* dari *node*. Himpunan *bridge-block* didapatkan dengan cara menelusuri arah *bridge-block parent* dari *node* hingga *bridge-block parent* dari sebuah *node* adalah *node* itu sendiri. Agar lebih optimal dilakukan juga *path* compresssion pada struktur data *disjoint set* yang merepresentasikan himpunan *bridge-block* dengan cara menghubungkan langsung setiap *node* yang dikunjungi dalam proses ini dengan *root* dari himpunan *bridge-block*. Pseudocode Fungsi FindBridgeBlock ditunjukkan pada Gambar 3.4

```
1  if u == u.bParent
2      return u
3  else u.bParent = FindBridgeBlock(u.bParent)
4      return u.bParent
```

Gambar 3.4: Pseudocode Fungsi FindBridgeBlock

3.5 Desain Fungsi FindComponent

Fungsi FindComponent akan memberikan nilai kembali berupa himpunan *connected component* dari *node*. Himpunan *connected component* didapatkan dengan cara menelusuri arah *connected component parent* dari *node* hingga *bridge-block parent* dari sebuah *node* adalah *node* itu sendiri. Pada fungsi ini juga dilakukan *path* compresssion pada struktur data *disjoint set* yang merepresentasikan himpunan *connected component*, setiap *node* yang dilewati pada akhirnya akan merujuk ke *root* dari *connected component*. Pseudocode Fungsi FindComponent ditunjukkan pada Gambar 3.5

```
1   uB = FindBridgeBlock(u)
2   if uB == uB.cParent
3       return uB
4   else uB.cParent = FindBridgeBlock(uB.cParent)
5       return uB.cParent
```

Gambar 3.5: Pseudocode Fungsi FindComponent

3.6 Desain Fungsi SetRoot

Fungsi SetRoot akan menjadikan *node* tersebut sebagai *root* dari himpunan *connected component* dengan cara mengubah arah *parent* pointer mengarah ke child berlaku untuk setiap *node* yang dikunjungi antara *path node* tersebut dengan *root*. Pseudocode Fungsi SetRoot ditunjukkan pada Gambar 3.6

```
1   now = FindBridgeBlock(u)
2   root = u
3   child = NULL
4   while now ≠ NULL
5       if now.parent ≠ NULL
6           next = FindBridgeBlock(now.parent)
7       else next = NULL
8       now.parent = child
9       now.cParent = root
10      child = now
11      now = next
12  root.cSize = child.cSize
```

Gambar 3.6: Pseudocode Fungsi SetRoot

3.7 Desain Fungsi MergeBridgeBlock

Fungsi MergeBridgeBlock menerima dua parameter u dan v yang merupakan *vertex* yang mengakibatkan terbentuknya *cycle* akibat penambahan *edge* di antara kedua *vertex* tersebut. Fungsi MergeBridgeBlock akan menggabungkan *bridge-block* yang terdapat dalam *path*(u, v) menjadi satu *bridge-block*. Proses menggabungkan *bridge-block* ini akan menghapus *bridge* yang berada di antara *path*(u, v). Penggabungan *bridge-block* dilakukan dengan cara mencari LCA dari *bridge-block* yang ada dalam *path*(u, v) dan seluruh himpunan *bridge-block* yang dikunjungi pada akhirnya akan merujuk pada LCA. Jumlah *bridge* pada proses ini akan berkurang sebanyak jumlah *bridge-block* yang dilewati dalam *path*(u, v) kecuali LCA. Pseudocode Fungsi MergeBridgeBlock ditunjukkan pada Gambar 3.7

```
1  lca = LCA(u, v)
2  for each node ∈ path(u, v)
3      if node ≠ lca
4          node.bParent = lca
5          bridge = bridge − 1
```

Gambar 3.7: Pseudocode Fungsi MergeBridgeBlock

BAB 4

IMPLEMENTASI

Pada bab ini akan dijelaskan implementasi dari algoritma dan struktur data berdasarkan desain yang telah dilakukan.

4.1 Lingkungan Implementasi

Lingkungan implementasi dan pengembangan yang dilakukan adalah sebagai berikut:

1. Perangkat Keras
   - Processor Intel Core i3-2120 CPU @ 3.30GHz x 2.
   - *Memory* 7.7 GB.
2. Perangkat Lunak
   - Sistem operasi Linux Mint 17.3 Rosa Cinnamon 64-bit.
   - Integrated Development Environment *Code*::Blocks 13.12.

4.2 Implementasi Fungsi Main

Fungsi Main diimplementasikan sesuai pseudocode subbab 3.1. Pada awalnya sistem akan mendeklarasikan variabel-variabel yang dibutuhkan dan melakukan inisialisasi nilai dari struct *node* yang merupakan representasi *vertex* pada graf. Untuk setiap masukan penambahan *edge* terhadap pasangan *vertex* sistem akan menjalankan fungsi InsertEdge(u, v) yang akan melakukan penambahan *edge* dan komputasi jumlah *bridge* lalu menampilkan luaran berupa jumlah *bridge* yang terdapat pada graf. Didefenisikan juga makro scanInt sebagai metode input yang lebih cepat dari scanf. Implementasi dari Fungsi Main dapat dilihat pada Kode Sumber 4.1.

```
1  #define scanInt(a) do c=getchar_unlocked();\
2  while(c<48); \
3  for(a=0;c>=48;c=getchar_unlocked())a=a*10+c-48;
4
5  int main() {
6      int t, m, u, v; char c;
7      labelCounter = 0;
8      struct Node node[50000];
9      scanInt(t);
10     while(t--) {
11         bridge = 0;
12         scanInt(n); scanInt(m);
13         for(int i=0; i<=n-1; i++) {
14             node[i].reset();
15         }
16         for(int i=1; i<=m; i++) {
17             scanInt(u); scanInt(v);
18             node[u].insertEdge(&node[v]);
19             printf("%d\n", bridge);
20         }
21     }
22     return 0;
23 }
```

Kode Sumber 4.1: Implementasi Fungsi Main

4.3 Implementasi Variabel Global

```
1  int n, bridge, labelCounter;
```

Kode Sumber 4.2: Implementasi Variabel Global

Variabel dengan *scope* global dibutuhkan untuk kemudahan pengaksesan sebuah variabel oleh setiap fungsi yang ingin mengaksesnya. Penggunaan variabel global pada implementasi ini diterapkan pada variabel *bridge* yang merepresentasikan jumlah *bridge* graf, variabel n yang merepresentasikan jumlah *vertex* pada graf dan juga

variabel labelCounter yang akan digunakan untuk pelabelan *vertex* saat mencari LCA dari dua bride-*block* dalam satu himpunan *connected component*. Implementasi dari variabel global dapat dilihat pada Kode Sumber 4.2.

4.4 Implementasi Struct *Node*

Struct *Node* merupakan implementasi dari desain Kelas *Node* pada subbab 3.2 dan merupakan representasi sebuah *vertex* v pada graf. Pada Struct *Node* juga terdapat implementasi algoritma yang digunakan pada komputasi jumlah *bridge*. Implementasi dari Struct *Node* dapat dilihat pada Kode Sumber 4.3.

```
1  struct Node {
2      struct Node *parent, *bParent, *cParent;
3      int cSize, label;
4
5      void reset() {
6          parent = NULL;
7          bParent = this;
8          cParent = this;
9          cSize = 1;
10         label = -1;
11     }
12 }
```

Kode Sumber 4.3: Implementasi Struct *Node*

Pada *line* 2 dilakukan deklarasi tipe data pointer yang akan disimpan pada struct *node*. Pointer *Node* *parent akan menyimpan nilai pointer dari *parent node*, pointer *bParent akan menyimpan alamat pointer *bridge-block parent* dari *node* dan pointer *cParent akan menyimpan alamat pointer *connected component parent* dari *node*. Pada *line* 3 dilakukan deklarasi nilai cSize dan nilai label. Nilai cSize adalah ukuran himpunan *connected component* tergabung dengan *vertex*. Sedangkan nilai label akan digunakan sebagai counter saat penggabungan *node* ketika penambahan *edge* menghasilkan

cycle.

4.5 Implementasi Fungsi InsertEdge

Fungsi InsertEdge melakukan proses penambahan *edge*(u, v) dengan memperhatikan beberapa kondisi seperti yang telah dijelaskan pada subbab 2.2. Implementasi dari Fungsi InsertEdge dapat dilihat pada Kode Sumber 4.4.

```
1  void insertEdge(struct Node *v) {
2      struct Node *bu = this->findBridgeBlock();
3      struct Node *bv = v->findBridgeBlock();
4      if(bu != bv) {
5          struct Node *cu = bu->findComponent();
6          struct Node *cv = bv->findComponent();
7          if(cu == cv) {
8              bu->mergeBridgeBlock(bv);
9          }
10         else {
11             if(cu->cSize < cv->cSize) {
12                 bu->setRoot();
13                 bu->parent = bv;
14                 bu->cParent = bv;
15                 cv->cSize += bu->cSize;
16             }
17             else {
18                 bv->setRoot();
19                 bv->parent = bu;
20                 bv->cParent = bu;
21                 cu->cSize += bv->cSize;
22             }
23             bridge++;
24         }
25     }
26 }
```

Kode Sumber 4.4: Implementasi Fungsi InsertEdge

Pada Kode Sumber 4.4 *line* 2-3 akan mendapatkan himpunan

bridge-block dari u dan v. Jika u dan v memiliki *bridge-block* yang sama maka tidak ada perubahan pada jumlah *bridge* maupun struktur data pada graf. Jika u dan v berada pada *bridge-block* yang berbeda maka akan dilihat nilai dari *connected component* masing-masing *node* u dan v seperti yang dijelaskan pada *line* 5-6. Jika u dan v terdapat pada himpunan *connected component* yang sama maka akan dipanggil fungsi mergeBridgeBlock. Jika u dan v berada pada himpunan *connected component* yang berbeda maka *edge* tersebut adalah *bridge* dan salah satu dari u atau v akan menjadi *root* dari himpunan *connected* componentnya dan menjadi child dari yang lain seperti yang dijelaskan pada *line* 11-23.

4.6 Implementasi Fungsi FindBridgeBlock

Fungsi FindBridgeBlock memberikan nilai kembali berupa himpunan *bridge-block* dari *vertex* tersebut. Implementasi dari Fungsi FindBridgeBlock dapat dilihat pada Kode Sumber 4.5.

```
1  struct Node* findBridgeBlock() {
2      struct Node* bParent = this->bParent;
3      if(bParent == this) return this;
4      else {
5          this->bParent = bParent->findBridgeBlock();
6          return this->bParent;
7      }
8  }
```

Kode Sumber 4.5: Implementasi Fungsi FindBridgeBlock

Kode Sumber 4.5 merupakan implementasi dari *path-compression disjoint-set* pada *bridge-block*. Jika bParent dari struct *node* adalah struct *node* itu sendiri maka fungsi akan memberikan nilai kembali *node* tersebut. Jika tidak maka fungsi akan mencari lagi secara rekursif dan pointer bParent dari setiap *node* yang dikunjungi akan mengarah pada *root* dari *bridge-block* tersebut.

4.7 Implementasi Fungsi FindComponent

Fungsi FindComponent memberikan nilai kembali berupa himpunan *connected component* dari *vertex* tersebut. Implementasi dari Fungsi FindComponent dapat dilihat pada Kode Sumber 4.6.

```
1  struct Node* findComponent() {
2      struct Node* b = this->findBridgeBlock();
3      if(b == b->cParent) return b;
4      else {
5          b->cParent = b->cParent->findComponent();
6          return b->cParent;
7      }
8  }
```

Kode Sumber 4.6: Implementasi Fungsi FindComponent

4.8 Implementasi Fungsi SetRoot

Fungsi SetRoot akan menjadikan *vertex* tersebut sebagai *root* pada himpunan *connected component*. Implementasi dari Fungsi SetRoot dapat dilihat pada Kode Sumber 4.7.

```
1  void setRoot() {
2      struct Node *now = this->findBridgeBlock();
3      struct Node *root = this, *child = NULL;
4      while(now) {
5          struct Node *b;
6          if(now->parent)
7              b = now->parent->findBridgeBlock();
8          else b = NULL;
9          now->parent = child;
10         now->cParent = root;
11         child = now;
12         now = b;
13     }
14     root->cSize = child->cSize;
15 }
```

Kode Sumber 4.7: Implementasi Fungsi SetRoot

4.9 Implementasi Fungsi MergeBridgeBlock

Fungsi MergeBridgeBlock menggabungkan *bridge-block* yang ada pada *path*(u, v) menjadi satu. Implementasi dari Fungsi MergeBridgeBlock dapat dilihat pada Kode Sumber 4.8.

```
1  void mergeBridgeBlock(struct Node *v) {
2      labelCounter++;
3      struct Node *pathX[50000], *pathY[50000];
4      int xEnd = 0, yEnd = 0;
5      struct Node *lca = NULL, *x = this, *y = v;
6      while(true) {
7          if(x) {
8              x = x->findBridgeBlock();
9              pathX[xEnd++] = x;
10             if (x->label == labelCounter) {
11                 lca = x; break;
12             }
13             x->label = labelCounter;
14             x = x->parent;
15         }
16         if(y) {
17             y = y->findBridgeBlock();
18             pathY[yEnd++] = y;
19             if (y->label == labelCounter) {
20                 lca = y; break;
21             }
22             y->label = labelCounter;
23             y = y->parent;
24         }
25     }
26     for(int i=0; i<xEnd && pathX[i] != lca; i++) {
27         pathX[i]->bParent = lca; bridge--;
28     }
29     for(int i=0; i<yEnd && pathY[i] != lca; i++) {
30         pathY[i]->bParent = lca; bridge--;
31     }
32 }
```

Kode Sumber 4.8: Implementasi Fungsi MergeBridgeBlock

Pada fungsi ini akan dilakukan pencarian LCA antara *bridge-block* dari u dan v. Karena u dan v berada pada himpunan *connected component* yang sama sebelumnya, maka *edge* yang ditambahkan akan membentuk *cycle* dan *edge* yang ada pada *path*(u, v) bukan lagi sebuah *bridge*. Untuk mendapatkan *bridge* yang ada pada *path*(u, v) ditelusuri *bridge-block parent* dimulai dari *vertex* u dan v. Pencarian LCA dilakukan dengan cara memberi label setiap *bridge-block* yang dikunjungi dan jika terdapat *node* yang menjadi titik pertemuan dari penelusuran *bridge-block vertex* u dan v maka *node* tersebut adalah LCA, proses ini dijelaskan pada *line* 6-27. *Line* 28-39 akan mengarahkan bParent dari setiap *node* yang telah dikunjungi untuk mengarah ke LCA yang telah didapatkan pada proses sebelumnya. Pada proses ini juga akan dilakukan pengurangan nilai *bridge* pada graf.

BAB 5

UJI COBA DAN EVALUASI

Pada bab ini akan dijelaskan tentang uji coba dan evaluasi dari implementasi sistem yang telah dilakukan pada bab 4.

5.1 Lingkungan Uji Coba

Lingkungan uji coba menggunakan sebuah komputer dengan spesifikasi perangkat lunak dan perangkat keras sebagai berikut:

- Perangkat Keras:
  - Processor Intel Core i3-2120 CPU @ 3.30GHz x 2.
  - *Memory* 7.7 GB.
  - 64-bit Operating System, x64 based processor.
- Perangkat Lunak:
  - Sistem operasi Linux Mint 17.3 Rosa Cinnamon 64-bit.
  - Integrated Development Environment *Code*::Blocks 13.12.

5.2 Skenario Uji Coba

Pada subbab ini akan dijelaskan skenario uji coba yang dilakukan. Skenario uji coba terdiri dari uji coba kebenaran dan uji coba kinerja.

5.2.1 Uji Coba Kebenaran

Uji coba kebenaran dilakukan dengan analisis penyelesaian sebuah contoh kasus dengan pendekatan penyelesaian yang telah dijelaskan pada subbab 2.2 dengan hasil dari luaran program dan pengumpulan

berkas kode sumber hasil implementasi ke situs sistem penilaian daring SPOJ. Permasalahan yang diselesaikan adalah *Online Bridge Searching* dengan kode ONBRIDGE.

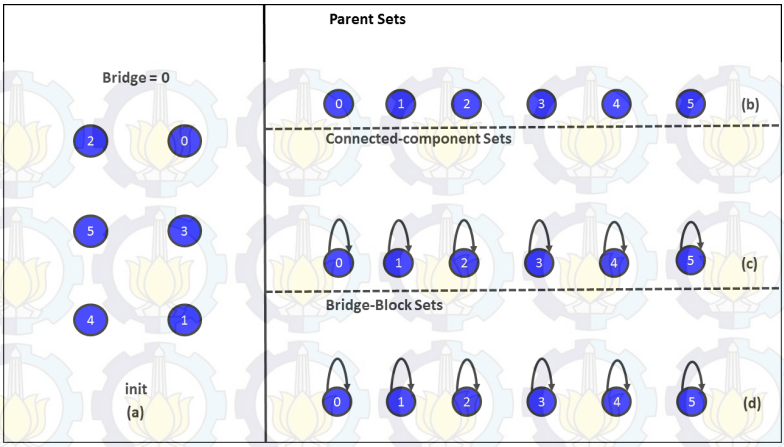
Kasus yang akan digunakan sebagai bahan uji kebenaran dalam analisis penyelesaian dengan hasil luaran program adalah data uji yang dihasilkan oleh data generator yang telah dijelaskan pada subbab 2.6. Hasil data uji dari data generator dengan jumlah 1 kasus uji, jumlah *vertex* 6 dan jumlah operasi penambahan *edge* 8 dapat dilihat pada Gambar 5.1.

```
1
6 8
5 3
5 4
1 4
2 5
4 2
3 2
5 1
1 3
```

Gambar 5.1: Contoh kasus uji hasil dari data generator.

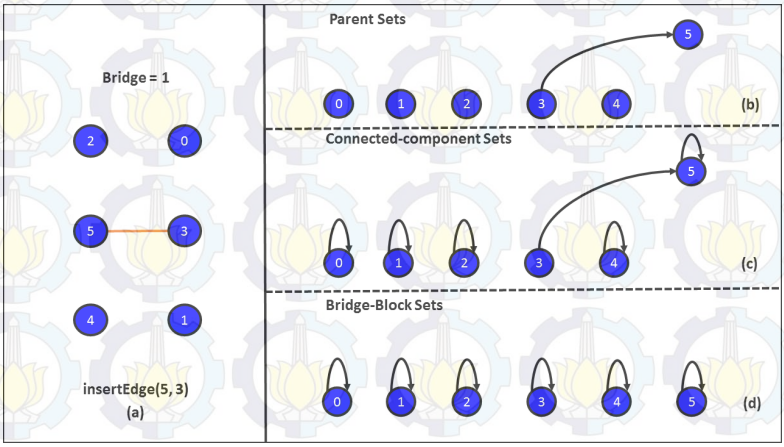
Langkah pertama dilakukan inisialisasi *node* dari sejumlah *vertex* yang ada pada kasus uji. Tahap inisialisasi dapat dilihat pada Gambar 5.2.

Vertex yang mengalami pertambahan *edge* pada operasi pertama adalah *vertex* 5 dan 3. Operasi ini membentuk *bridge* dan karena kedua *vertex* memiliki ukuran himpunan *connected-component* yang sama maka salah satu *vertex* (*vertex* 5) menjadi *parent* dari *vertex* lainnya (*vertex* 3) pada *parent* sets dan *connected-component* sets. Pada operasi ini jumlah *bridge* bertambah menjadi 1 sehingga program seharusnya mengeluarkan luaran 1 setelah operasi ini. Ilus-



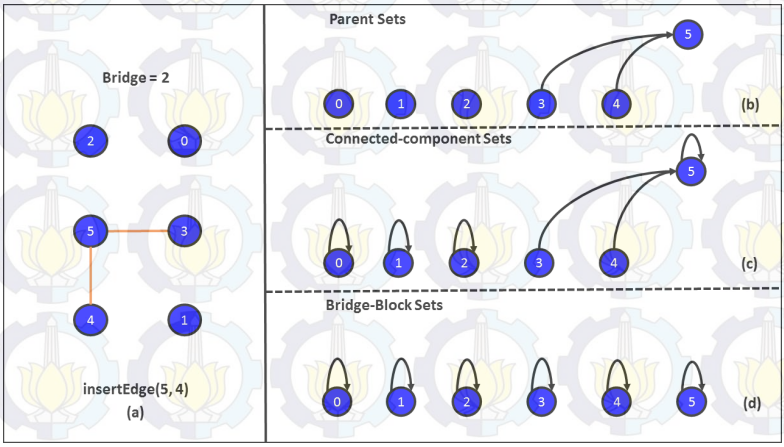
Gambar 5.2: Inisialisasi contoh kasus uji.

trasi dari operasi penambahan *edge*(5, 3) dijelaskan pada Gambar 5.3.

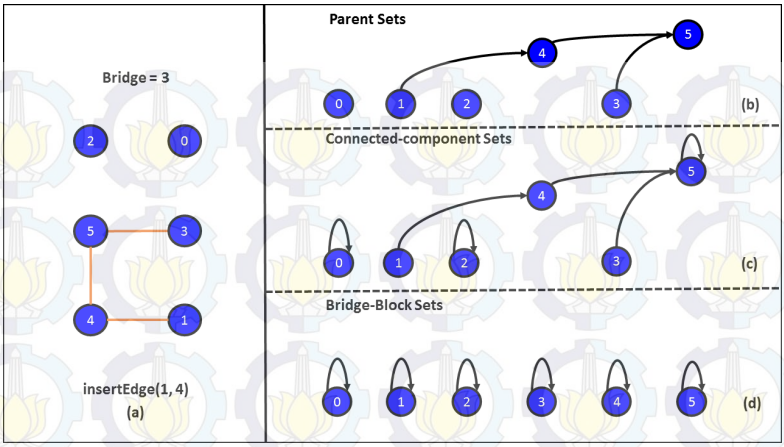


Gambar 5.3: Ilustrasi penambahan *edge*(5, 3) pada contoh kasus uji.

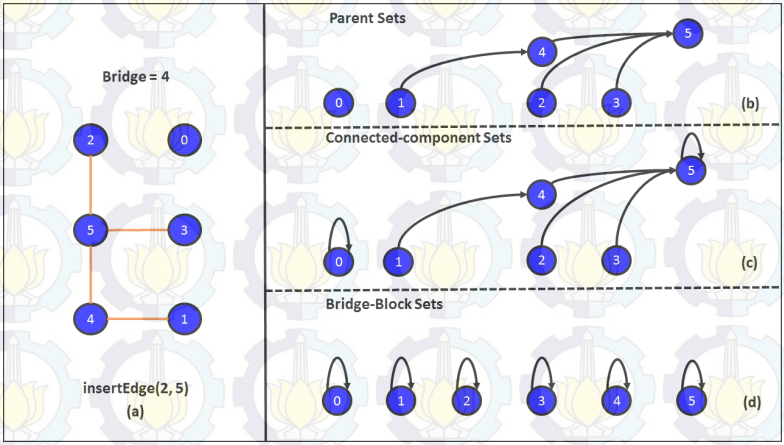
Operasi selanjutnya menambahkan *edge* antara *vertex* 5 dan *vertex* 4. Ukuran himpunan *connected-component* dari *vertex* 5 saat ini lebih besar daripada *vertex* 4. Sehingga *vertex* 4 akan menjadikan dirinya *root* dari himpunan *connected*-componentnya dan *parent* dari *vertex* 4 selanjutnya adalah *bridge-block root* dari *vertex* 5 yang mana merupakan *vertex* 5 itu sendiri. Setelah operasi ini jumlah *bridge* pada graf bertambah menjadi 2. Operasi penambahan *edge* antara *vertex* 1 dan 4 juga membentuk *bridge* karena 1 memiliki ukuran himpunan *connected-component* yang lebih kecil maka 1 akan menjadi *root* dari himpunan *connected*-componentnya dan *parent* dari 1 selanjutnya adalah *vertex* 4. Jumlah *bridge* setelah operasi ini adalah 3. Penambahan *edge* antara *vertex* 2 dan 5 membentuk *bridge* dan *parent* dari *vertex* 2 pada *component*-sets dan *parent*-sets adalah *vertex* 5. Jumlah *bridge* setelah operasi ini menjadi 4. Ilustrasi dari ketiga operasi di atas dijelaskan pada Gambar 5.4, 5.5 dan 5.6.



Gambar 5.4: Ilustrasi penambahan *edge*(5, 4) pada contoh kasus uji.



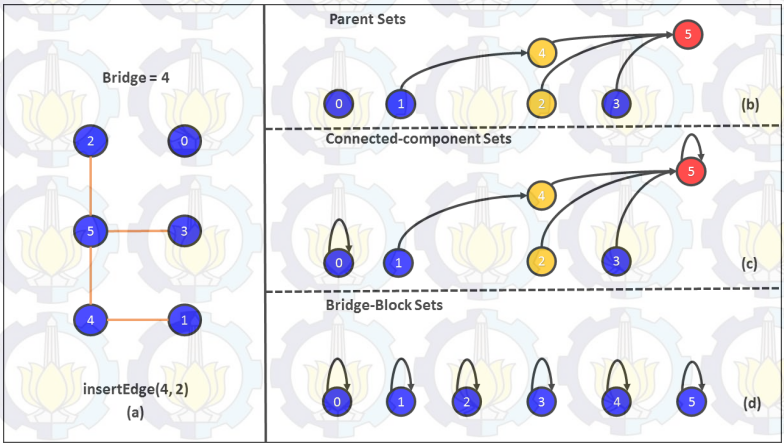
Gambar 5.5: Ilustrasi penambahan *edge*(1, 4) pada contoh kasus uji.



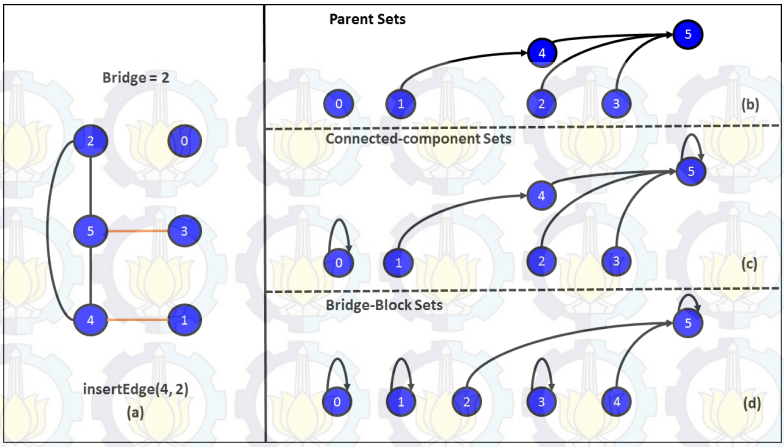
Gambar 5.6: Ilustrasi penambahan *edge*(2, 5) pada contoh kasus uji.

Operasi penambahan *edge* antara *vertex* 4 dan 2 akan membentuk *cycle* dikarenakan *vertex* 4 dan 2 berada pada himpunan *connected-component* yang sama namun dalam *bridge-block* yang berbeda.

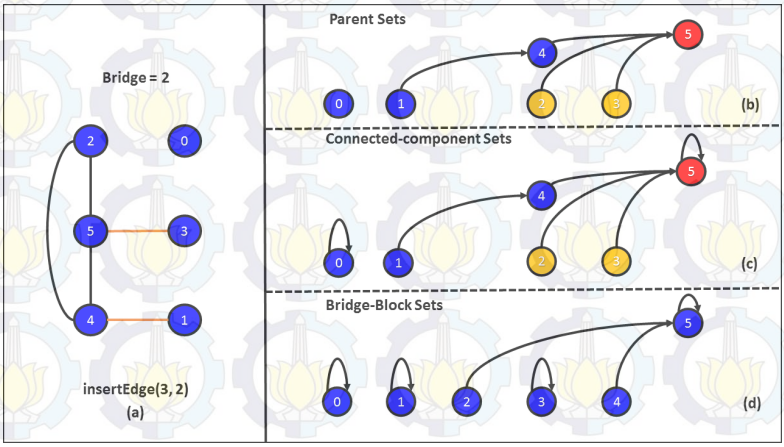
Maka akan dicari LCA antara kedua *vertex* dengan cara masing-masing *vertex* mencari *bridge-block root* masing-masing dan kemudian menuju pada *bridge-block* selanjutnya dengan cara melihat *parent* dari *vertex* pada *parent* sets. Akhirnya didapatkan LCA antara kedua *vertex* tersebut adalah *vertex* 5 dan semua *bridge-block* yang telah dilewati sebelumnya (*vertex* 4 dan *vertex* 2) akan merujuk pada *vertex* 5 pada *bridge-block* sets. *Edge* yang dilewati selama proses ini bukan lagi sebuah *bridge-block* sehingga jumlah *bridge* setelah operasi ini adalah 2. Operasi penambahan *edge* antara *vertex* 3 dan 2 juga membentuk *cycle*. LCA antara kedua *vertex* ini adalah *vertex* 5. Karena *bridge-block root* dari *vertex* 2 adalah 5 maka jumlah *bridge* yang berkurang pada operasi ini hanya 1 dari *vertex* 3 ke 5. Selanjutnya *vertex* 3 akan merujuk pada *vertex* 5 pada *bridge-block* sets. Jumlah *bridge* setelah operasi ini adalah 1. Kedua operasi ini diilustrasikan pada Gambar 5.7, 5.8, 5.9 dan 5.10.



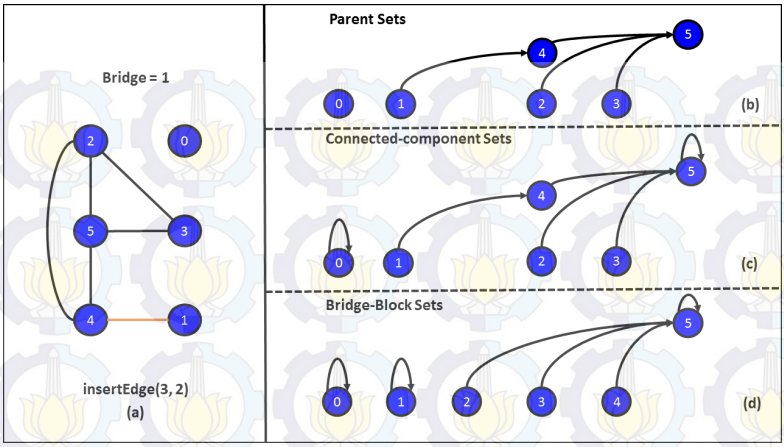
Gambar 5.7: Ilustrasi penambahan *edge*(4, 2) pada contoh kasus uji (1).



Gambar 5.8: Ilustrasi penambahan *edge*(4, 2) pada contoh kasus uji (2).



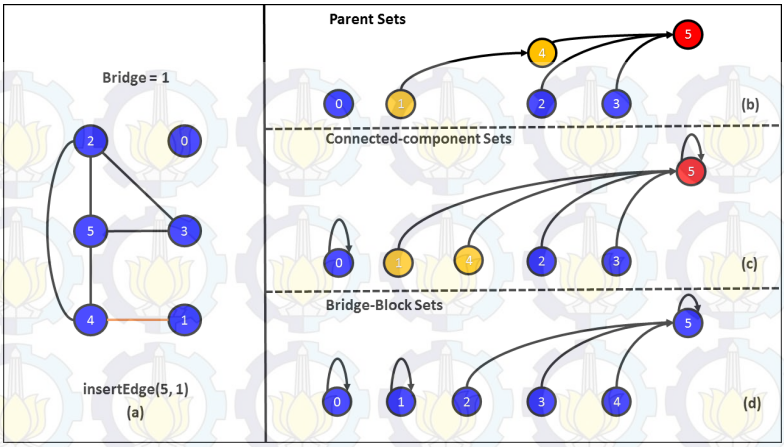
Gambar 5.9: Ilustrasi penambahan *edge*(3, 2) pada contoh kasus uji (1).



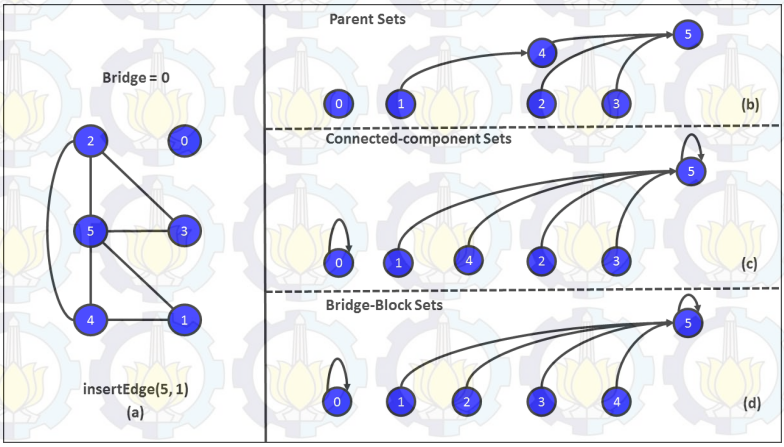
Gambar 5.10: Ilustrasi penambahan *edge*(3, 2) pada contoh kasus uji (2).

Operasi selanjutnya adalah operasi penambahan *edge* antara *vertex* 5 dan 1. Operasi ini membentuk *cycle* maka *edge* yang ada di antara *path*(1, 5) bukan lagi sebuah *bridge*. *Bridge-block* dan *edge* yang berada di antara *path*(1, 5) bisa didapatkan dengan mencari LCA antara kedua *vertex* dengan cara melihat ke arah *bridge-block root* dari *vertex* lalu melihat ke arah *parent* dari *vertex* pada himpunan *parent* sets dan berulang hingga mencapai titik temu antara kedua *vertex*. LCA antara kedua *vertex* ini adalah *vertex* 5 dan jumlah *bridge* yang berkurang adalah 1 dari *vertex* 1 ke *vertex* 4. Selanjutnya *vertex* 1 akan merujuk pada *vertex* 5 pada *bridge-block* sets dan jumlah *bridge* setelah operasi ini adalah 0. Operasi ini digambarkan pada Gambar 5.11 dan 5.12.

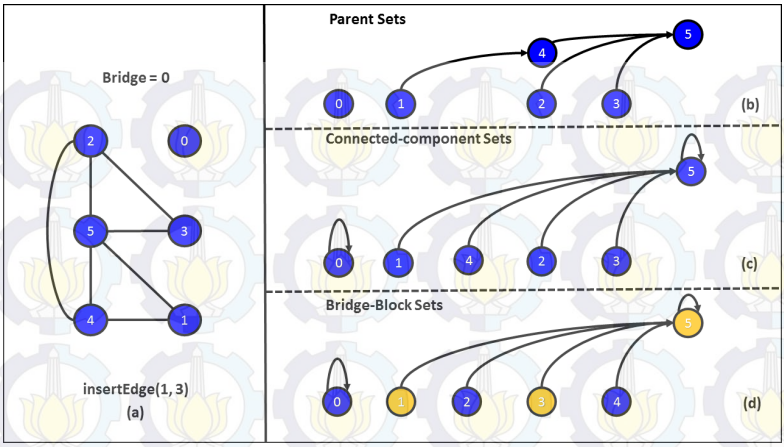
Operasi terakhir yaitu penambahan *edge* antara *vertex* 1 dan 3 tidak merubah apapun dari struktur data karena *vertex* 1 dan 3 sudah berada pada *bridge-block* dan juga tidak merubah jumlah *bridge* pada graf. Operasi ini digambarkan pada Gambar 5.13 dan 5.14.



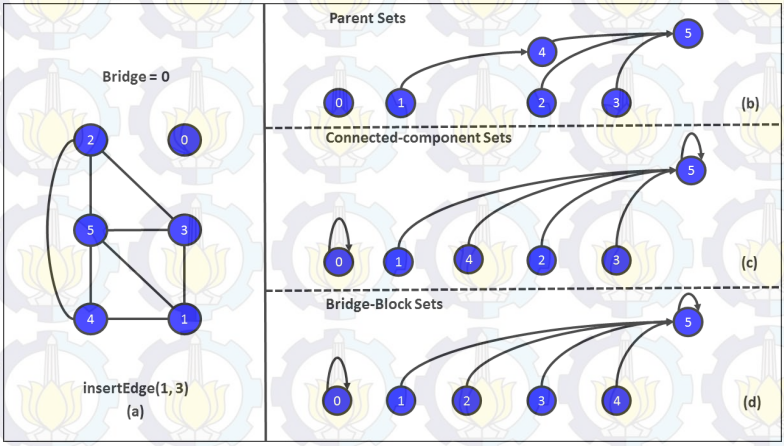
Gambar 5.11: Ilustrasi penambahan *edge*(5, 1) pada contoh kasus uji (1).



Gambar 5.12: Ilustrasi penambahan *edge*(5, 1) pada contoh kasus uji (2).



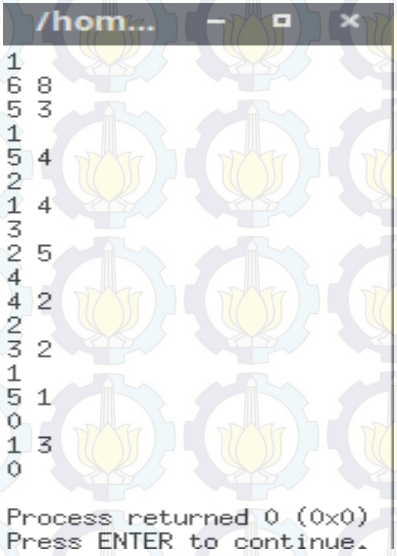
Gambar 5.13: Ilustrasi penambahan *edge*(1, 3) pada contoh kasus uji (1).



Gambar 5.14: Ilustrasi penambahan *edge*(1, 3) pada contoh kasus uji (2).

Dari hasil analisis di atas urutan jumlah *bridge* untuk setiap operasi adalah 1, 2, 3 4, 2, 1, 0 dan 0. Kemudian sistem penyelesaian dijalankan dan diberi masukan sesuai dengan contoh kasus uji dari

analisis sebelumnya dan hasil dari luaran sistem adalah 1, 2, 3 ,4, 2, 1, 0 dan 0 seperti yang terlihat pada Gambar 5.15.



Gambar 5.15: Hasil luaran program pada contoh kasus uji.

Selanjutnya dilakukan juga uji kebenaran dengan mengumpulkan berkas kode sumber ke situs SPOJ dapat dilihat umpan balik sistem. Hasil uji kebenaran dan peringkat waktu eksekusi program saat pengumpulan kasus uji pada situs SPOJ ditunjukkan pada Gambar 5.16 dan 5.17.

Gambar 5.16: Hasil uji kebenaran pada situs SPOJ.

Dari hasil uji coba yang telah dilakukan kode sumber program mendapat umpan balik *Accepted*. Waktu yang dibutuhkan program adalah 0.02 detik dan memori yang dibutuhkan program adalah 3.9 MB.

RANK	DATE	USER	RESULT	TIME	MEM	LANG
1	2015-12-01 08:45:49	Yusro Tsaqova	accepted	0.02	3.9M	C++ 4.3.2
2	2013-06-01 12:11:43	Bidhan	accepted	0.04	3.7M	C++ 4.3.2
3	2014-02-18 20:49:46	Aristofanis Rontogiannis	accepted	0.04	5.7M	C++ 5

Gambar 5.17: Peringkat waktu eksekusi program *Online Bridge Searching* pada situs SPOJ.

Hal itu membuktikkan bahwa implementasi yang dilakukan telah berhasil menyelesaikan permasalahan komputasi jumlah *bridge* sesuai dengan batasan-batasan yang telah ditetapkan. Setelah itu dilakukan pengiriman kode sumber implementasi sebanyak 30 kali untuk melihat variasi waktu dan memori yang dibutuhkan program. Grafik hasil uji coba sebanyak 30 kali ditunjukkan dalam Gambar 5.18.

Dari hasil pengumpulan kode sebanyak 30 kali, didapat waktu rata-rata program yaitu 0.031 detik dan penggunaan memori rata-rata yang dibutuhkan program yaitu 3.9MB.

5.2.2 Uji Coba Kinerja

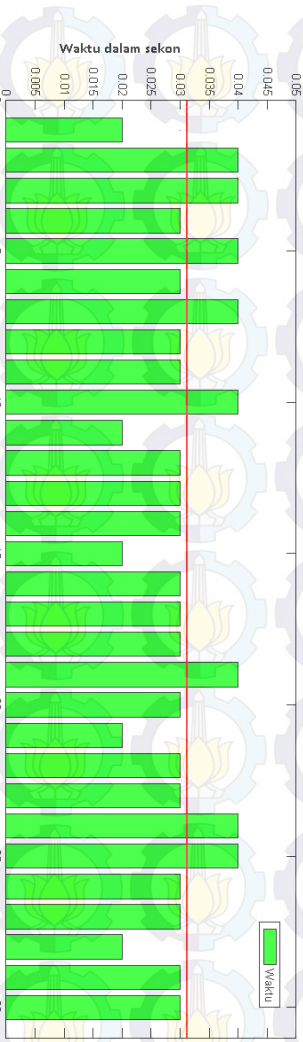
Untuk n *vertex* dan m operasi penambahan *edge*, algoritma yang ditawarkan memiliki kompleksitas waktu $\mathcal{O}(M(\alpha(N)))$ Uji coba kinerja dilakukan dengan membuat operasi penambahan *edge* pada graf secara acak sesuai dengan batasan masalah pada subbab 2.5. Setelah itu dilakukan komparasi kinerja untuk melihat pengaruh jumlah *vertex* dan pengaruh operasi penambahan *edge* terhadap pertumbuhan waktu.

5.2.2.1 Pengaruh Jumlah *Vertex* Terhadap Waktu

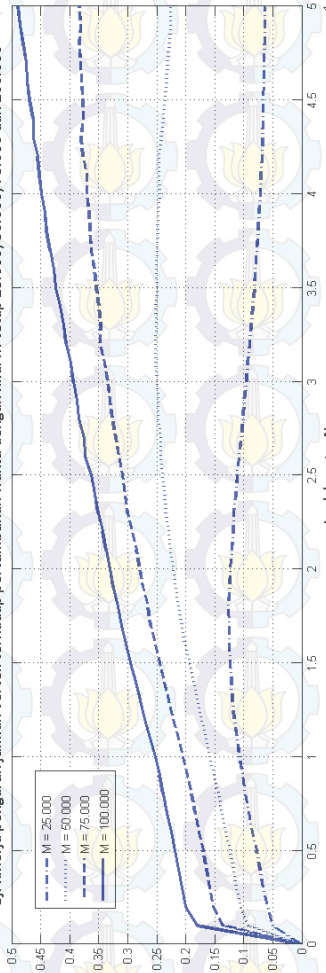
Jumlah *vertex* diatur bervariasi antara 1.000 hingga 50.000 *vertex* dengan rentang 1.000 dan banyak operasi penambahan *edge* dibuat tetap yaitu 25.000, 50.000, 75.000 dan 100.000 untuk melihat perbandingan pengaruh jumlah *vertex* terhadap pertumbuhan waktu dengan jumlah operasi tetap. Hasil uji coba ditunjukkan pada Gambar 5.19.

5.2.2.2 Pengaruh Jumlah Operasi Terhadap Waktu

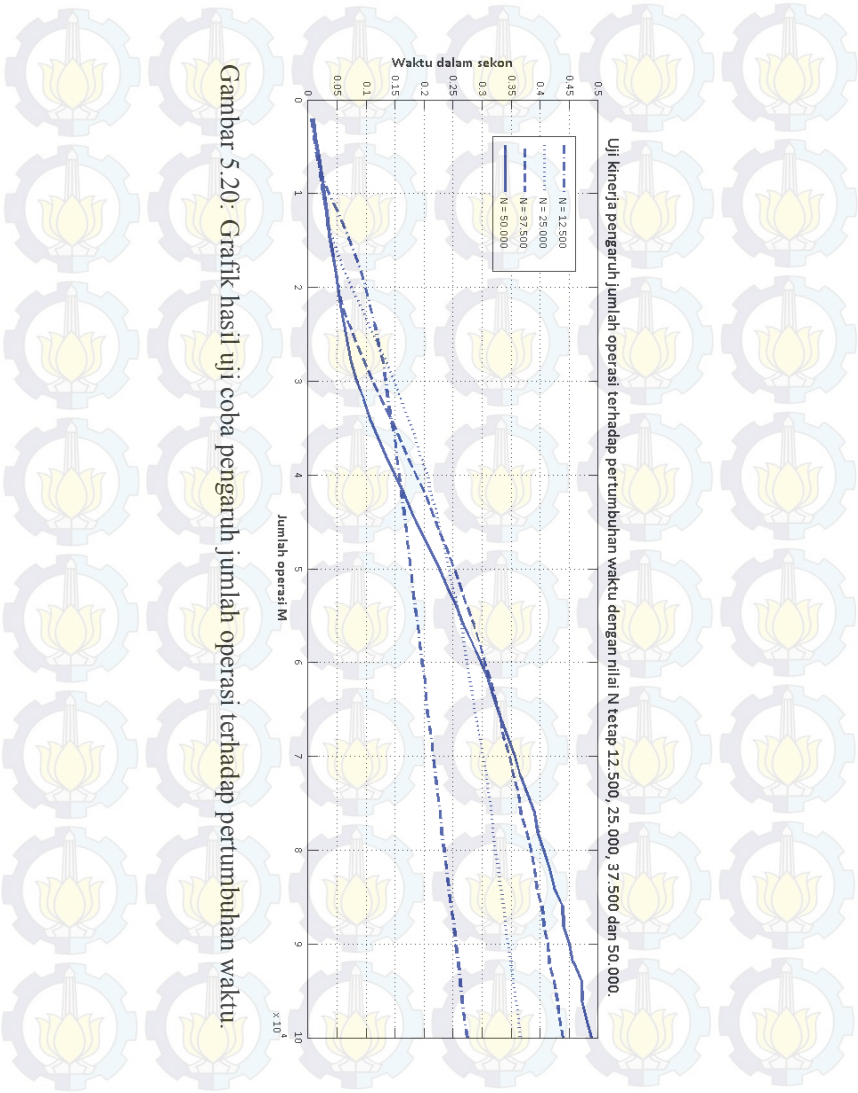
Jumlah operasi penambahan *edge* diatur bervariasi antara 2.000 hingga 100.000 *vertex* dengan rentang 2.000 dan banyak *vertex* dibuat tetap yaitu 12.500, 25.000, 37.500 dan 50.000 untuk melihat perbandingan pengaruh jumlah operasi penambahan *edge* terhadap pertumbuhan waktu dengan jumlah *vertex* tetap. Hasil uji coba ditunjukkan pada Gambar 5.20.



Gambar 5.18: Grafik Hasil uji pada situs SPOJ sebanyak 30 kali.



Gambar 5.19: Grafik hasil uji coba pengaruh jumlah *vertex* terhadap pertumbuhan waktu.



Gambar 5.20: Grafik hasil uji coba pengaruh jumlah operasi terhadap pertumbuhan waktu.

BAB 6

KESIMPULAN

Pada bab ini akan dijelaskan kesimpulan dari hasil uji coba yang telah dilakukan.

6.1 Kesimpulan

Dari hasil uji coba yang telah dilakukan terhadap implementasi solusi untuk permasalahan komputasi jumlah *bridge* pada graf dinamis inkremental dapat diambil kesimpulan sebagai berikut:

1. Penyelesaian permasalahan komputasi jumlah *bridge* pada graf dinamis inkremental dengan memperhatikan kondisi pasangan *vertex* yang mengalami pertambahan *edge* dengan bantuan struktur data *disjoint set* dan pendekatan heuristik *union by weight* dan *path-compression* memiliki kompleksitas waktu *amortized* $\mathcal{O}(M(\alpha(N)))$ dimana $\alpha(N)$ adalah invers dari *fast-growing ackermann function* dan bernilai ≤ 4 untuk setiap N.
2. Informasi himpunan *bridge-block* dan *connected-component* yang terkait pada sebuah *vertex* dapat dimanfaatkan untuk komputasi jumlah *bridge* secara *online* ketika ada modifikasi pada graf karena pertambahan *edge*.
3. Implementasi dari solusi yang ditawarkan pada Tugas Akhir ini telah berhasil menyelesaikan permasalahan *Online Bridge Searching* sesuai dengan batasan yang telah ditentukan dengan cukup optimal.
4. Waktu dan penggunaan memori rata-rata yang diperlukan untuk menyelesaikan permasalahan *Online Bridge Searching*

dari 30 kali uji coba pada situs SPOJ adalah 0.031 detik dan penggunaan memori sebesar 3.9 MB.

DAFTAR PUSTAKA

[1] D. Eppstein, Z. Galil, and G. F. Italiano. ***Dynamic graph* algorithms**, CRC Press, 1997

[2] H. T. Cormen, E. C. Leiserson, L. R. Rivest and C. Stein, **Introduction to Algorithms Third Edition**, Cambridge, Massachusetts: The MIT Press, 2009

[3] J. Westbrook, R. E. *Tarjan*. **Maintaining *Bridge-Connected* and Biconnected Components On-*line***, CRC Press, 1997

[4] *Tarjan*, Robert E. **Efficiency of a Good But Not Linear *Set Union Algorithm***, Journal of the ACM, 22(2):215–225, 1975

[5] M. A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena and P. Sumazin. **Lowest Conommon Ancestors in Trees and *Directed* Acyclic Graphs**, ACM Press, 2001

[6] SPOJ, **ONBRIDGE - *Online Bridge Searching***, [*Online*], `http://www.spoj.com/problems/ONBRIDGE/en/`, diakses pada tanggal 22 Desember 2015

Halaman ini sengaja dikosongkan

LAMPIRAN A

HASIL UJI PADA SITUS SPOJ SEBANYAK 30 KALI

Berikut merupakan lampiran hasil uji coba pengumpulan berkas kode solusi pada situs SPOJ sebanyak 30 kali dari grafik yang ditampilkan pada Gambar 5.18.

15752027	2015-12-01 11:10:11	Online Bridge Searching	accepted edit ideone it	0.04	3.9M	C++ 4.3.2
15752012	2015-12-01 11:07:30	Online Bridge Searching	accepted edit ideone it	0.03	3.9M	C++ 4.3.2
15752006	2015-12-01 11:06:44	Online Bridge Searching	accepted edit ideone it	0.03	3.9M	C++ 4.3.2
15752000	2015-12-01 11:05:57	Online Bridge Searching	accepted edit ideone it	0.04	3.9M	C++ 4.3.2
15751998	2015-12-01 11:05:43	Online Bridge Searching	accepted edit ideone it	0.03	3.9M	C++ 4.3.2
15751997	2015-12-01 11:05:30	Online Bridge Searching	accepted edit ideone it	0.04	3.9M	C++ 4.3.2
15751994	2015-12-01 11:05:15	Online Bridge Searching	accepted edit ideone it	0.03	3.9M	C++ 4.3.2
15751992	2015-12-01 11:04:55	Online Bridge Searching	accepted edit ideone it	0.04	3.9M	C++ 4.3.2
15751987	2015-12-01 11:04:06	Online Bridge Searching	accepted edit ideone it	0.04	3.9M	C++ 4.3.2
15751261	2015-12-01 09:19:50	Online Bridge Searching	accepted edit ideone it	0.02	3.9M	C++ 4.3.2

Gambar A.1: Hasil uji pada situs SPOJ sebanyak 30 kali (1).

15912115	2015-12-21 15:49:16	Online Bridge Searching	accepted	0.03	3.9M	C++ 4.3.2
15912112	2015-12-21 15:48:55	Online Bridge Searching	accepted	0.03	3.9M	C++ 4.3.2
15912110	2015-12-21 15:48:35	Online Bridge Searching	accepted	0.02	3.9M	C++ 4.3.2
15912106	2015-12-21 15:48:12	Online Bridge Searching	accepted	0.03	3.9M	C++ 4.3.2
15912103	2015-12-21 15:47:55	Online Bridge Searching	accepted	0.03	3.9M	C++ 4.3.2
15912101	2015-12-21 15:47:33	Online Bridge Searching	accepted	0.04	3.9M	C++ 4.3.2
15912094	2015-12-21 15:46:54	Online Bridge Searching	accepted	0.04	3.9M	C++ 4.3.2
15912091	2015-12-21 15:46:29	Online Bridge Searching	accepted	0.03	3.9M	C++ 4.3.2
15912088	2015-12-21 15:46:03	Online Bridge Searching	accepted	0.03	3.9M	C++ 4.3.2
15912084	2015-12-21 15:45:42	Online Bridge Searching	accepted	0.02	3.9M	C++ 4.3.2

Gambar A.2: Hasil uji pada situs SPOJ sebanyak 30 kali (2).

15912079	2015-12-21 15:45:26	Online Bridge Searching	accepted	0.03	3.9M	C++ 4.3.2
15912073	2015-12-21 15:45:11	Online Bridge Searching	accepted	0.04	3.9M	C++ 4.3.2
15912063	2015-12-21 15:44:12	Online Bridge Searching	accepted	0.03	3.9M	C++ 4.3.2
15912061	2015-12-21 15:43:53	Online Bridge Searching	accepted	0.03	3.9M	C++ 4.3.2
15912056	2015-12-21 15:43:07	Online Bridge Searching	accepted	0.03	3.9M	C++ 4.3.2
15912055	2015-12-21 15:42:54	Online Bridge Searching	accepted	0.02	3.9M	C++ 4.3.2
15912052	2015-12-21 15:42:38	Online Bridge Searching	accepted	0.03	3.9M	C++ 4.3.2
15912050	2015-12-21 15:42:24	Online Bridge Searching	accepted	0.03	3.9M	C++ 4.3.2
15912047	2015-12-21 15:42:10	Online Bridge Searching	accepted	0.03	3.9M	C++ 4.3.2
15912044	2015-12-21 15:41:51	Online Bridge Searching	accepted	0.02	3.9M	C++ 4.3.2

Gambar A.3: Hasil uji pada situs SPOJ sebanyak 30 kali (3).

Halaman ini sengaja dikosongkan

LAMPIRAN B

HASIL UJI KINERJA JUMLAH OPERASI M TETAP

Berikut merupakan lampiran hasil uji kinerja dengan nilai jumlah operasi M tetap dari grafik pada Gambar 5.19.

Tabel B.1: Hasil uji dengan nilai jumlah operasi M tetap 25.000.

No.	Jumlah *Vertex*	Jumlah Operasi	Waktu dalam sekon
1	1000	25000	0.05
2	2000	25000	0.059
3	3000	25000	0.066
4	4000	25000	0.072
5	5000	25000	0.078
6	6000	25000	0.085
7	7000	25000	0.091
8	8000	25000	0.097
9	9000	25000	0.102
10	10000	25000	0.107
11	11000	25000	0.111
12	12000	25000	0.116
13	13000	25000	0.119
14	14000	25000	0.121
15	15000	25000	0.123
16	16000	25000	0.125
17	17000	25000	0.125
18	18000	25000	0.126
19	19000	25000	0.124
20	20000	25000	0.123
21	21000	25000	0.12

Tabel B.1: Hasil uji dengan nilai jumlah operasi M tetap 25.000 (lanjutan).

No.	Jumlah *Vertex*	Jumlah Operasi	Waktu dalam sekon
22	22000	25000	0.118
23	23000	25000	0.117
24	24000	25000	0.113
25	25000	25000	0.111
26	26000	25000	0.108
27	27000	25000	0.104
28	28000	25000	0.101
29	29000	25000	0.099
30	30000	25000	0.095
31	31000	25000	0.093
32	32000	25000	0.089
33	33000	25000	0.087
34	34000	25000	0.084
35	35000	25000	0.081
36	36000	25000	0.08
37	37000	25000	0.077
38	38000	25000	0.075
39	39000	25000	0.073
40	40000	25000	0.071
41	41000	25000	0.07
42	42000	25000	0.068
43	43000	25000	0.067
44	44000	25000	0.066
45	45000	25000	0.066
46	46000	25000	0.065
47	47000	25000	0.065
48	48000	25000	0.064
49	49000	25000	0.064
50	50000	25000	0.064

Tabel B.2: Hasil uji dengan nilai jumlah operasi M tetap 50.000.

No.	Jumlah *Vertex*	Jumlah Operasi	Waktu dalam sekon
1	1000	50000	0.092
2	2000	50000	0.105
3	3000	50000	0.112
4	4000	50000	0.118
5	5000	50000	0.124
6	6000	50000	0.131
7	7000	50000	0.137
8	8000	50000	0.144
9	9000	50000	0.151
10	10000	50000	0.158
11	11000	50000	0.165
12	12000	50000	0.172
13	13000	50000	0.18
14	14000	50000	0.186
15	15000	50000	0.193
16	16000	50000	0.201
17	17000	50000	0.207
18	18000	50000	0.212
19	19000	50000	0.217
20	20000	50000	0.221
21	21000	50000	0.226
22	22000	50000	0.229
23	23000	50000	0.234
24	24000	50000	0.236
25	25000	50000	0.24
26	26000	50000	0.243
27	27000	50000	0.243
28	28000	50000	0.247
29	29000	50000	0.247
30	30000	50000	0.249

Tabel B.2: Hasil uji dengan nilai jumlah operasi M tetap 50.000 (lanjutan).

No.	Jumlah *Vertex*	Jumlah Operasi	Waktu dalam sekon
31	31000	50000	0.251
32	32000	50000	0.252
33	33000	50000	0.25
34	34000	50000	0.251
35	35000	50000	0.249
36	36000	50000	0.25
37	37000	50000	0.249
38	38000	50000	0.248
39	39000	50000	0.247
40	40000	50000	0.245
41	41000	50000	0.246
42	42000	50000	0.246
43	43000	50000	0.242
44	44000	50000	0.238
45	45000	50000	0.235
46	46000	50000	0.233
47	47000	50000	0.231
48	48000	50000	0.228
49	49000	50000	0.226
50	50000	50000	0.226

Tabel B.3: Hasil uji dengan nilai jumlah operasi M tetap 75.000.

No.	Jumlah *Vertex*	Jumlah Operasi	Waktu dalam sekon
1	1000	75000	0.135
2	2000	75000	0.152
3	3000	75000	0.158
4	4000	75000	0.164
5	5000	75000	0.17
6	6000	75000	0.177
7	7000	75000	0.184
8	8000	75000	0.19
9	9000	75000	0.197
10	10000	75000	0.204
11	11000	75000	0.213
12	12000	75000	0.221
13	13000	75000	0.23
14	14000	75000	0.237
15	15000	75000	0.244
16	16000	75000	0.252
17	17000	75000	0.26
18	18000	75000	0.265
19	19000	75000	0.273
20	20000	75000	0.279
21	21000	75000	0.285
22	22000	75000	0.292
23	23000	75000	0.3
24	24000	75000	0.304
25	25000	75000	0.309
26	26000	75000	0.315
27	27000	75000	0.32
28	28000	75000	0.326
29	29000	75000	0.33
30	30000	75000	0.334

Tabel B.3: Hasil uji dengan nilai jumlah operasi M tetap 75.000 (lanjutan).

No.	Jumlah *Vertex*	Jumlah Operasi	Waktu dalam sekon
31	31000	75000	0.339
32	32000	75000	0.347
33	33000	75000	0.346
34	34000	75000	0.349
35	35000	75000	0.354
36	36000	75000	0.357
37	37000	75000	0.362
38	38000	75000	0.365
39	39000	75000	0.366
40	40000	75000	0.37
41	41000	75000	0.37
42	42000	75000	0.377
43	43000	75000	0.38
44	44000	75000	0.376
45	45000	75000	0.377
46	46000	75000	0.379
47	47000	75000	0.381
48	48000	75000	0.383
49	49000	75000	0.381
50	50000	75000	0.383

Tabel B.4: Hasil uji dengan nilai jumlah operasi M tetap 100.000.

No.	Jumlah *Vertex*	Jumlah Operasi	Waktu dalam sekon
1	1000	100000	0.18
2	2000	100000	0.2
3	3000	100000	0.206
4	4000	100000	0.212
5	5000	100000	0.218
6	6000	100000	0.225
7	7000	100000	0.232
8	8000	100000	0.239
9	9000	100000	0.245
10	10000	100000	0.252
11	11000	100000	0.261
12	12000	100000	0.27
13	13000	100000	0.279
14	14000	100000	0.286
15	15000	100000	0.295
16	16000	100000	0.303
17	17000	100000	0.31
18	18000	100000	0.317
19	19000	100000	0.325
20	20000	100000	0.33
21	21000	100000	0.337
22	22000	100000	0.343
23	23000	100000	0.351
24	24000	100000	0.356
25	25000	100000	0.362
26	26000	100000	0.373
27	27000	100000	0.375
28	28000	100000	0.384
29	29000	100000	0.387
30	30000	100000	0.394

Tabel B.4: Hasil uji dengan nilai jumlah operasi M tetap 100.000 (lanjutan).

No.	Jumlah *Vertex*	Jumlah Operasi	Waktu dalam sekon
31	31000	100000	0.398
32	32000	100000	0.406
33	33000	100000	0.41
34	34000	100000	0.416
35	35000	100000	0.424
36	36000	100000	0.427
37	37000	100000	0.434
38	38000	100000	0.44
39	39000	100000	0.442
40	40000	100000	0.448
41	41000	100000	0.452
42	42000	100000	0.455
43	43000	100000	0.463
44	44000	100000	0.463
45	45000	100000	0.469
46	46000	100000	0.473
47	47000	100000	0.474
48	48000	100000	0.48
49	49000	100000	0.485
50	50000	100000	0.489

LAMPIRAN C

HASIL UJI KINERJA JUMLAH *VERTEX* N TETAP

Berikut merupakan lampiran hasil uji kinerja dengan nilai jumlah *vertex* N tetap dari grafik pada Gambar 5.20.

Tabel C.1: Hasil uji dengan nilai jumlah *vertex* N tetap 12.500.

No.	Jumlah *Vertex*	Jumlah Operasi	Waktu dalam sekon
1	12500	2000	0.005
2	12500	4000	0.01
3	12500	6000	0.015
4	12500	8000	0.023
5	12500	10000	0.036
6	12500	12000	0.051
7	12500	14000	0.064
8	12500	16000	0.077
9	12500	18000	0.089
10	12500	20000	0.099
11	12500	22000	0.107
12	12500	24000	0.115
13	12500	26000	0.121
14	12500	28000	0.129
15	12500	30000	0.133
16	12500	32000	0.138
17	12500	34000	0.142
18	12500	36000	0.149
19	12500	38000	0.151
20	12500	40000	0.157
21	12500	42000	0.161

Tabel C.1: Hasil uji dengan nilai jumlah *vertex* N tetap 12.500 (lanjutan).

No.	Jumlah *Vertex*	Jumlah Operasi	Waktu dalam sekon
22	12500	44000	0.164
23	12500	46000	0.169
24	12500	48000	0.172
25	12500	50000	0.177
26	12500	52000	0.181
27	12500	54000	0.185
28	12500	56000	0.189
29	12500	58000	0.194
30	12500	60000	0.196
31	12500	62000	0.202
32	12500	64000	0.203
33	12500	66000	0.208
34	12500	68000	0.212
35	12500	70000	0.215
36	12500	72000	0.22
37	12500	74000	0.223
38	12500	76000	0.227
39	12500	78000	0.231
40	12500	80000	0.234
41	12500	82000	0.238
42	12500	84000	0.242
43	12500	86000	0.246
44	12500	88000	0.251
45	12500	90000	0.254
46	12500	92000	0.259
47	12500	94000	0.262
48	12500	96000	0.265
49	12500	98000	0.269
50	12500	100000	0.274

Tabel C.2: Hasil uji dengan nilai jumlah *vertex* N tetap 25.000.

No.	Jumlah *Vertex*	Jumlah Operasi	Waktu dalam sekon
1	25000	2000	0.007
2	25000	4000	0.011
3	25000	6000	0.016
4	25000	8000	0.021
5	25000	10000	0.026
6	25000	12000	0.031
7	25000	14000	0.038
8	25000	16000	0.048
9	25000	18000	0.06
10	25000	20000	0.074
11	25000	22000	0.09
12	25000	24000	0.105
13	25000	26000	0.119
14	25000	28000	0.135
15	25000	30000	0.147
16	25000	32000	0.16
17	25000	34000	0.172
18	25000	36000	0.185
19	25000	38000	0.193
20	25000	40000	0.205
21	25000	42000	0.211
22	25000	44000	0.221
23	25000	46000	0.228
24	25000	48000	0.236
25	25000	50000	0.243
26	25000	52000	0.249
27	25000	54000	0.257
28	25000	56000	0.265
29	25000	58000	0.268
30	25000	60000	0.273

Tabel C.2: Hasil uji dengan nilai jumlah *vertex* N tetap 25.000 (lanjutan).

No.	Jumlah *Vertex*	Jumlah Operasi	Waktu dalam sekon
31	25000	62000	0.279
32	25000	64000	0.284
33	25000	66000	0.289
34	25000	68000	0.296
35	25000	70000	0.301
36	25000	72000	0.307
37	25000	74000	0.31
38	25000	76000	0.316
39	25000	78000	0.318
40	25000	80000	0.322
41	25000	82000	0.328
42	25000	84000	0.331
43	25000	86000	0.337
44	25000	88000	0.339
45	25000	90000	0.344
46	25000	92000	0.349
47	25000	94000	0.353
48	25000	96000	0.358
49	25000	98000	0.362
50	25000	100000	0.365

Tabel C.3: Hasil uji dengan nilai jumlah *vertex* N tetap 37.500.

No.	Jumlah *Vertex*	Jumlah Operasi	Waktu dalam sekon
1	37500	2000	0.008
2	37500	4000	0.012
3	37500	6000	0.017
4	37500	8000	0.021
5	37500	10000	0.026
6	37500	12000	0.031
7	37500	14000	0.035
8	37500	16000	0.041
9	37500	18000	0.046
10	37500	20000	0.052
11	37500	22000	0.059
12	37500	24000	0.07
13	37500	26000	0.083
14	37500	28000	0.097
15	37500	30000	0.111
16	37500	32000	0.126
17	37500	34000	0.143
18	37500	36000	0.156
19	37500	38000	0.171
20	37500	40000	0.185
21	37500	42000	0.201
22	37500	44000	0.213
23	37500	46000	0.226
24	37500	48000	0.24
25	37500	50000	0.251
26	37500	52000	0.263
27	37500	54000	0.273
28	37500	56000	0.285
29	37500	58000	0.296
30	37500	60000	0.304

Tabel C.3: Hasil uji dengan nilai jumlah *vertex* N tetap 37.500 (lanjutan).

No.	Jumlah *Vertex*	Jumlah Operasi	Waktu dalam sekon
31	37500	62000	0.315
32	37500	64000	0.323
33	37500	66000	0.332
34	37500	68000	0.336
35	37500	70000	0.347
36	37500	72000	0.353
37	37500	74000	0.362
38	37500	76000	0.366
39	37500	78000	0.376
40	37500	80000	0.384
41	37500	82000	0.39
42	37500	84000	0.394
43	37500	86000	0.403
44	37500	88000	0.406
45	37500	90000	0.413
46	37500	92000	0.415
47	37500	94000	0.424
48	37500	96000	0.429
49	37500	98000	0.433
50	37500	100000	0.439

Tabel C.4: Hasil uji dengan nilai jumlah *vertex* N tetap 50.000.

No.	Jumlah *Vertex*	Jumlah Operasi	Waktu dalam sekon
1	50000	2000	0.009
2	50000	4000	0.013
3	50000	6000	0.018
4	50000	8000	0.023
5	50000	10000	0.027
6	50000	12000	0.032
7	50000	14000	0.036
8	50000	16000	0.041
9	50000	18000	0.046
10	50000	20000	0.051
11	50000	22000	0.056
12	50000	24000	0.062
13	50000	26000	0.068
14	50000	28000	0.074
15	50000	30000	0.083
16	50000	32000	0.096
17	50000	34000	0.106
18	50000	36000	0.12
19	50000	38000	0.134
20	50000	40000	0.149
21	50000	42000	0.164
22	50000	44000	0.178
23	50000	46000	0.194
24	50000	48000	0.21
25	50000	50000	0.226
26	50000	52000	0.239
27	50000	54000	0.255
28	50000	56000	0.267
29	50000	58000	0.283
30	50000	60000	0.297

Tabel C.4: Hasil uji dengan nilai jumlah *vertex* N tetap 50.000 (lanjutan).

No.	Jumlah *Vertex*	Jumlah Operasi	Waktu dalam sekon
31	50000	62000	0.311
32	50000	64000	0.321
33	50000	66000	0.332
34	50000	68000	0.344
35	50000	70000	0.356
36	50000	72000	0.365
37	50000	74000	0.378
38	50000	76000	0.39
39	50000	78000	0.395
40	50000	80000	0.405
41	50000	82000	0.416
42	50000	84000	0.424
43	50000	86000	0.438
44	50000	88000	0.44
45	50000	90000	0.45
46	50000	92000	0.457
47	50000	94000	0.471
48	50000	96000	0.472
49	50000	98000	0.479
50	50000	100000	0.488

BIODATA PENULIS

Yusro Tsaqova, lahir di Mataram tanggal 3 Juni 1994. Penulis merupakan anak pertama dari 2 bersaudara. Penulis telah menempuh pendidikan formal TK Perwanida I Ampenan, SD Negeri 37 Ampenan (2000-2006), SMP Negeri 2 Mataram (2006-2009) dan SMA Negeri 1 Mataram (2009-2012). Penulis melanjutkan studi kuliah program sarjana di Jurusan Teknik Informatika ITS. Selama masa kuliah, penulis aktif dalam organisasi Himpunan Mahasiswa Teknik Computer-Informatika (HMTC) ITS. Penulis juga aktif dalam kegiatan kepanitiaan Schematics sebagai Staff National Programming Contest (NPC) Schematics 2013 dan Ketua NPC Schematics 2014. Selain itu penulis juga aktif menjadi administrator Lab Pemrograman (LP) Teknik Informatika ITS.

Selama kuliah di Teknik Informatika ITS, penulis mengambil bidang minat Dasar dan Terapan Komputasi (DTK). Penulis pernah menjadi asisten dosen dan praktikum untuk mata kuliah Pemrograman Terstruktur (2013 dan 2014), Algoritma dan Struktur data (2013), Struktur Data (2014) dan Dasar Pemrograman (2015). Selama menempuh perkuliahan penulis juga aktif mengikut kompetisi pemrograman tingkat nasional dan menjadi finalis kategori pemrograman pada lomba COMPFEST UI (2013, 2014 dan 2015), INC Bina Nusantara (2013, 2014 dan 2015) dan ICPC Regional Asia-Jakarta (2013, 2014 dan 2015). Selain itu penulis juga pernah menjadi asisten Pelatnas 2 TOKI (2015). Penulis dapat dihubungi melalui surel di `yusrotsaqova@gmail.com`.